GetErrors(IN TimeCriteria, IN ErrorType, OUT ErrorList)

This is called by a server UI.

The admin UI program can query the ADRM server for any errors. The errors can be categorized by type of errors or errors that occur between certain time periods. A small sample of the possible ADRM server errors includes: client access token timeout, failure to read user information from the account database, failure to get the license information, failure to write usage information into the database, etc…

GetIllegalAccesses(IN TimeCriteria, IN AccessType, OUT AccessList)

This is called by a server UI.

The admin UI program can query the ADRM server for any illegal accesses. The illegal accesses can be categorized by type and time period. A small sample of the possible ADRM server errors includes: failure attempts to access ADRM server with bad password repeatedly in a small time period, failure attempts to use a particular license, any access attempts from a non-typical IP address ranges for a particular account, etc…

GetAppID(IN AppName, OUT AppID)

This is called by a server UI.

Returns a unique identifier associated a particular application

SetApp(ID IN AppName, IN AppID)

This is called by a server UI.

Stores a unique identifier associated a particular application

Application Server

Purpose

The application server is there to handle read requests for files accessed by eStream clients. Any file accessed on a client through the EFS can have this read request passed to an app server.

Functionality

This will be the hardest working eStream server. It will respond to both synchronous (demand fetching) and asynchronous (prefetching) page requests from many different clients, for many different types of applications and files within those applications.

Interfaces

GetFileInfo(IN AccessToken, IN FileID, OUT FileInfo)

This is called from an eStream client. Given any file within an eStream application, return metadata about it. The access token is provided for validation.

ReadFile(IN AccessToken, IN FileID, IN Length, IN Offset, OUT Buffer, OUT BytesRead)

This interface is called by an eStream client, and will allow the client to access any eStreamed application file and AppInstallBlocks. How the FileID for an AppInstallBlock is achieved is unclear at present.

OpenFile() / GetFileID()

Note: This is a placeholder for an API that may be needed. This depends a lot on the eventual communications between client and server for associating a file pathname with a FileID.

ASP web server

Purpose

This describes, of course, only those interfaces on an ASP web server that relate to handling eStreamed applications.

Logically, the ASP web server is the backend web interface for user requests—e.g., get billing information, subscribe to a new app, or request a list of all possible apps a user can subscribe to. In the current model, the web server doesn't actually handle these requests, but instead passes them on to the appropriate eStream-centric server.

NOTE: The following interfaces are not updated from the previous version! They were written with the assumption that the web server actually manages all the data described above. We need the server team to suggest the changes that should take place here!

Functionality

Interfaces

AddADRMServer()

The ASP Web Server is informed of the availability of a new ADRM server. The ASP Web Server adds this new ADRM Server to its list of online ADRM Servers. Periodically, the ASP Web Server can query the list of ADRM Servers for its load

information. When a new client connects to the ASP Web Server, the client can be informed of the subset of ADRM Servers with the least load.

Callers: ASP Web Server

Input:

- ADRM Server IP
- list of Account Ids that the ADRM Server supports.

Output:

- success or failure

RemoveADRMServer()

The ASP Web Server is told of the removal of an ADRM Server. It must remove the ADRM Server from it's locally cached list of ADRM Servers to prevent any future clients from using that particular ADRM Server.

Caller(s): ASP Web Server

Input:

- ADRM Server

Output:

- success or failure

ValidateSubscribedUser()

Inputs:

- SubscriptionToken

Outputs:

Validate subscribed user is called by the ADRM server to check out a license.

- Find account(accountNumber)
- Find user(username)
- Find license(SubscriptionID)

If a license is found to be available, check it out and return OK else return NO_LICENSE_AVAILABLE.

Add/RemoveAccount()

Input:

- ownerUsername
- ownerPasswd
- billingInfo

Output:

- accountNumber

Creator must supply info for the first user, who is also the account owner.

Add/RemoveSubscribableApp()

Input:

- application
- AppServer locations
- name description

AddAccountUser()

Input:

- account number
- user name
- initial password

Add/Remove/Increment/DecrementFloatingLicense()

Input:

- account number

- subscription
- number available

Add/RemovePerUserLicense()

Input:

- account number
- user
- subscription

CompileAccountUsage()

Input:

- account number

Add up and report all the usage by members of the account.

ClearAccountUsage()

FreeLicense()

Input:

- user
- subscription

Frees a license that had been previously checked-out by a user.

GetUserPriveleges()

Input:

- account number
- username

Check privileges of a logged in user for the purposes of allowing user/subscription management and other account changes

ShowCheckedoutLicenses()

GetAccountInfo()

Input:

- account number
- username

ListPossibleSubscriptions()

Input: none

ListCurrentSubscriptions()

Input:

- accountNumber

CreateSubscription()

Input:

- account number
- license data (depends on license type)
- ApplicationPackage

Output:

- subscriptionID
- ADRM server names.

Builder

Does not talk to any other module. Probably an associated set of tools managed via a script plus manual procedures that create intermediate data files, and finally produce the eStream set.

Farm Manager

Simply takes user commands and input, activating the following functions on the actual Application Server process:

StopServer(boolean graceful)

ConfigureServer(config_parameters)

EnableEstreamSet(appID) - informs the app server that the set is ready to be served

DisableEstreamSet(appID) - opposite of above

level stuff would be an extension of the above. Synchronizing the availability of apps between the Application Server and ADRM/Account DB must also be handled; at least initially, a human administrator should be able to flips the final switch.

Monitor

The monitor utility is responsible for monitoring the overall health of the system. It is responsible to report server status, server traffic, illegal access etc. It will ping the Application Server and the ADRM server to gather the statistics and display them.

Server data objects

This section needs work! What should be here?

eStream Set

What is an eStream set? It consists of:

- Page prediction map, indicates likelihood a page will be referenced successively after another page. Used to enable accurate prefetching by the client.
- Spoofer info, stuff to initialize register & file spoofer to enable application to run on the client. (ITARD & "Spoofed file mapping list")
- Application content. This includes all files of the application in possibly multiple subtrees (stuff from C:, Z:, Common Files, etc.). These are all then placed under a single application root directory which is "mounted" under the eStream file system under the application's directory ("Microsoft Office" or whatever). This directory structure is then processed to create a special eStream metadata file to represent it, and map file names to FileIDs, one assigned to each file for the app, used by the client. All of the application files can then be placed into a single large file with a FileID index in front to allow the **Application Server** to map requested FileIDs to offsets in the large application content image file.
- The above takes care of everything in the AppInstallBlock except for possible COM dlls that might be necessary (TBD what mechanism generates these).

Account/Subscription Database:

Description:

The Account Subscription database manages all the data required to manage accounts, users and subscription rights, to log usage, and compile billing for application rental. This document describes the data model for the db, the list of accessor APIs, and finally, a list of scenarios that exercise these APIs.

Data Model

The following is a description of the data model for the Account/Subscription Database.

There are two types of records described.

Static Objects are objects that are stored persistently in the Account/Subscription database. The attributes of each record are divided into two categories:

- Owns: specifies data that is actually contained in and managed by the object. These are the attributes that make the object what it is. For example: an account is not an account unless it has billing information. Each item (except the ASP) must have only one owner, but may have many references.
- References: Attributes that create associations to other first-class objects, making navigation from one item to another simpler.

Transient Objects represent data structures that are created in order to pass information from place to another. For the most part, they provide shorthand identifiers for static objects.

Each attribute listed below has the format:

Type: name – (optional) description

For attributes that don't yet have a defined type, the type is *undefined*

Static Data (Database records)

ASP – The top-level container.

Owns:

- List<Account>: accounts - all of the accounts
- List<Subscription>: subscriptions - all of the available subscriptions
- List<Applications>: applications - all of the applications currently served by Application Servers

References:

- None

Account - Collection of attributes that make up a single account:

Owns:

- Number: accountNumber – number that identifies this account
- List<User>: users – all of the account users
- Undefined: billingInfo – Currently undefined type

- List<License>: licenses - all of the licenses (per-user and floating) that are managed by the account
- List<UsageRecord> usage – list of records that describe usage of all account users

References:

- User: owner - a user who created the account – must be one of the users

User – a single user of an Account

Owns:

- String: username
- String: password
- Undefined: Role – specifies permissions on the account, i.e. owner, administrator vs. regular user
- Undefined: UserInfo – Real name, contact info etc

References:

- List<License>: licenses currently held by user
- List<License>: licenses - licenses to which the user has access. These might be either per-user or floating licenses, or a combination of the two. This list might also incorporate a means of specifying a preference – for example if a floating license and a per-user license are available, use the per-user

Application – single application that has been made available on one or more application servers.

Owns

- String: name
- String: description
- Undefined: AppServer location(s) – A location may be a host name that must be resolved by the appserver farm, or may be an IP address.
- Number: AppID
- A list of FileIDs for this app.

References:

- None

Subscription – An application or group of applications that have been made available for rental by a user.

Owns:

- Number: SubscriptionID
- String: name
- String: description

References:

- User: user
- List<Application>: applications - application(s) associated with this subscription

ApplicationPackage – *application(s) that can be rented*

Owns:

- Undefined pricing

References:

- List<Application> applications

License – base for other licenses – All licenses support the same APIs – check out, check in

Owns:

- None

References:

- Subscription: subscription – subscription for which this license grants rights

FloatingLicense – one of a fixed number of licenses distributed to a list of users on a first-come first-serve basis. Contains all attributes in License plus:

Owns:

- Number: numTotal

References:

- List<User> holders – the current holders of this license (length = numInUse)
- List<User> allowedUsers – list of users allowed to check out this license

PerUserLicense *– license tied to a particular user – one desktop (machine) at a time. Contains attributes in license plus:*

Owns:

- Boolean: isInUse
- Undefined – desktopID

References:

- User: allowedUser – the (only) user allowed to pull this license.

UsageRecord – Describes a billable use of the application

Owns:

- Undefined: Start time
- Undefined: End time

References:

- Subscription: subscription
- User: user

Transient Data

AccountNumber – integer that uniquely identifies a single account with an application service provider.

UserName – string that uniquely identifies a single user **within** an account. To uniquely specify a user to an ASP, it is necessary to qualify the UserName with the AccountNumber of which he is a member. All users have

UserVerifier – combination of the AccountNumber, UserName, and UserPassword. Uniquely identifies a user within an ASP

- AccountNumber
- UserName
- UserPassword

SubscriptionID – Integer that uniquely identifies a subscribable application or collection of applications within an ASP.

SubscriptionToken – describes a user-subscription to the ADRM server. Identifies the subscription as well as the user trying to access it.

- UserVerifier
- SubscriptionID

AppID – A unique numeric representation of each eStreamed application. For example, word := 1000, excel := 1001, office := 1002 etc. We should be able to represent software packages using AppIDs.

FileID – Within an eStreamed app, each app-file gets a unique numeric ID.

eStream 1.0 Requirements

Version 1.1

[redacted]

1.0 Introduction

This document describes the high level requirements for the eStream 1.0 product. These requirements are given first as lists for the client and server components and then as scenarios.

To facilitate the development of follow-on products, eStream 1.0 does not include attributes that explicitly preclude future support of thin clients or of data ubiquity.

2.0 Client Requirements

The following are performance and functional requirements for the client portion of eStream 1.0:

ID	Description	Priority
1.0	eStream client software operates on Windows 2000, Windows NT4, & Windows 98 for x86 clients.	10
1.1	eStream client software collects profile information during application execution; this information is used to improve client-based prefetching. The profile data may be uploaded [unless user disables this feature] to a server.	8
1.2	eStream client software supports eStream execution of top-selling (as represented by Ziff-Davis suites) desktop & laptop applications; these applications are listed in section 5.0 below. Other applications are supported (opportunistically) as well.	10
1.3	eStream client software is obtained via the web or via some distribution media & is installed via some industry-standard [e.g., installshield] mechanism; its installation requires administrative privileges.	10
	[illegible] minimize potential stability/reliability problems. eStream client software can be upgraded w/o reinstallation and w/o breaking installed apps.	
1.4	eStream client software operates across the different languages supported by Windows.	8
1.5	Applications running under eStream have an ave	8

	interactive response time within 10% of client native for connections at 256K bps or higher.	
1.6	eStream client software is able to operate with only 16M of available disk space; this is the minimum supported configuration. User is encouraged to use cache sized for best performance.	8
1.7	eStream client software supports simultaneous execution of multiple eStreamed applications, including multiple instances of a single application.	10
1.8	eStream client software is able to unambiguously reference a particular license for an application.	10
1.9	Applications being eStreamed function in the same way that they would if they were installed locally.	10
1.10	eStream client software tolerates server failure [i.e., it continues running any active apps and allowing apps to be launched], though possibly with some delay, assuming that an alternative server of the needed type is accessible.	10
1.11	eStream client software detects and tolerates lost or garbled messages.	10
1.12	It is difficult to steal an eStream application's code or data from the client.	10
1.13	When an eStream application is being installed on a client, the process detects if the app is already installed & requests user confirmation to continue; a single version of an application is available to the user at a time. Install reboot should be avoided, unless needed to minimize potential stability/reliability problems.	10
1.14	Upon uninstalling an application, all application-specific changes to the client system are removed or undone.	9~~10~~
1.15	eStream client software makes minimal changes to the client system when running, avoiding/hiding registry, DLL, & non-Z file system changes as needed.	10
1.16	eStream client software makes a run/no run license decision quickly enough when an eStreamed application is started not to cause customer satisfaction issues.	10
1.17	eStream client software is launched/terminated at boot/shutdown, at logon/logoff, or on demand.	10
1.18	User is able to set initial size of client cache & to increase the size of the cache later without significant performance penalty.	9
1.19	eStream client software does not include explicit	9

	ASP logon/logoff to run installed apps; ASPIDBlock stored on client machine gets AccessToken to execute app in seamless manner.	
1.20	eStream client software facilitates roaming (i.e., running one's eStream apps from a different client or moving one's client system to a different site).	10
1.21	eStream allows exporting information concerning installed apps on one client to one or more files which can be copied to another client. eStream server info is not invalid/inappropriate when the client is moved to a different venue.	8
1.22	eStream will not allow users to run the same application from multiple clients simultaneously if the license prohibits it.	10

3.0 Server Requirements

The following are performance and functional requirements for the server portion of eStream 1.0:

ID	Description	Priority
1.0	eStream provides user and account management capabilities.	10
1.1	User Account Creation/Deletion supported.	10
1.2	User Account is able to subscribe/unsubscribe to Applications.	10
1.3	User is able to view billing and account information.	10
1.4	User is able to change password/address/billing information online.	10
1.5	User is able to list all available & subscribed applications.	10
1.6	User is able to access online help/doc, including an FAQ database.	10
1.7	Omnishift provides interfaces to facilitate customer support by third parties.	10
1.8	User is able to enter/modify data securely.	10
1.9	Both IE 4.0/later and Netscape Navigator 4.0/later browsers are supported.	9
1.10	ASP agent [i.e., special administrative user at the ASP] is able to access all user information.	10
1.11	ASP agent is able to disable a user.	10

1.12	ASP agent is able to modify license information for a user.	10
1.13	User is able to add additional users to the account.	10
1.14	Web Server is highly scalable.	10
1.15	Servers are able to operate in non-English language.	9
1.16	ASP may operate eStream system with single server.	9
1.17	Flexible access/export of billing information is supported, to facilitate 3rd party billing systems.	10
1.18	eStream server software and eStream apps can be upgraded w/o impacting installed eStream client software. All upgrades are backwards compatible.	10
2.0	The eStream framework [ASLM Server] provides a mechanism to validate the usage of application components with respect to billing models.	10
2.1	ASLM ~~DRM~~ server is able to validate users to use specific applications.	10
2.2	ASLM~~DRM~~ server records all usage activity down to the granularity necessary to support billing models. The granularity will be reasonably large.	10
2.3	ASLM~~DRM~~ is able to release license on explicit request or timeout from the client.	10
2.4	ASLM~~DRM~~ is portable across a wide variety of platforms and operating systems, including but not limited to: Windows NT4, Windows 2000, Solaris UltraSPARC, and Linux.	10
2.5	ASLM~~DRM~~ servers are fault-tolerant.	10
2.6	ASLM~~DRM~~ servers are scalable.	10
2.7	ASLM~~DRM~~ server is able to report Denial of Service attempts.	10
2.8	ASLM~~DRM~~ server reports illegal accesses.	10
2.9	ASLM~~DRM~~ is able to register its presence/load to the Web Server(s).	9
3.0	eStream Framework provides management and monitoring tool (EMMT) to manage the servers.	10
3.1	EMMT is able to start/stop servers in the eStream framework.	10
3.2	EMMT is able to monitor server activity for all	10
3.3	EMMT is able to configure the servers.	10
3.4	EMMT is able to provide historical reporting.	9
3.5	EMMT is able to display information graphically and in spreadsheet format.	8
3.6	EMMT is able to raise alarms on predefined events.	9
4.0	eStream framework provides a mechanism to deploy the application via the eStream Builder.	10

5.0	eStream framework support a variety of licensing models.	10
5.1	Floating license model is supported. n User – k Licenses	10
5.2	Names User License model. (Special case n=k)	10
5.3	Time based licenses at billing granularity.	10
5.4	High water mark license.	10
5.5	Node locked licenses.	8
6.0	App Server is able to Authenticate client's accesses (via AccessTokens) completely locally.	10
6.1	App Server encrypts returned data (via a random key chosen by the client); it must be computationally infeasible to steal an application's code while it is being distributed or to determine which application a client is running.	10
6.2	App Server is as stateless as possible to allow client to switch to alternative app server w/o significant overhead. "Stateless" means that there is no server context that would be lost if the server went down; one classic example of this is that "file open" is recorded on the client, not on the server.	108
6.3	App Server is optimized to respond to requests with minimal server load, thereby maximizing scalability.	10
6.4	App Servers may be grouped along with any number of other such servers into a farm with minimal inter-server interactions (as to maximize scalability), with load balancing support.	9
6.5	App Server communicates with clients thru firewalls.	10
6.6	App Server communicates with clients efficiently (e.g., via persistent HTTP connections).	10
6.7	App Server is able to install new eStream sets w/o having to go down.	10
6.8	App Server is robust, able to run for long periods without crashes (i.e. no resource leaks, and handles most/all failure modes for system operations); 24/7 operation.	10
	EMMT communicate through a database which will include but need not be limited to Microsoft SQLServer.	

4.0 Builder Requirements

The following are performance and functional requirements for the builder portion of eStream 1.0:

ID	Description	Priority
1.0	The Builder installation monitor runs in the background, when an eStream application is installed as part of its preparation or building capabilities.	10
1.1	The Builder installation monitor captures all the updates to the System Registry that take place during the install.	10
1.2	The Builder installation monitor records all the files created in the two kinds of directories: the install directory and the common directories.	10
1.3	The Builder must be able to gather initial set of application profile data. This data consists at least of the page access pattern for starting and immediately shutting down an application	10
1.4	The Builder must package the eStream Set into an easily manageable packages suitable for ASP administrators to download to their servers.	10
1.5	The Builder must be able to collect per-user profile data from the Profile Server and merge the profile data into a combined data usable for updating the profile data in the appInstallBlock.	8
1.6	The Builder should be run in an environment where no other applications are running.	10
1.7	The Builder should provide functionality to create installation set(s) for each of the clients eStream 1.0 is going to support.	10
1.8	It should be possible to change the appId of the eStream set when an ASP wants to "install" the eStream set in order to host it.	10
1.9	It should be possible to create a merged eStream set for a suite of applications.	10
[illegible]	[illegible] by the Builder using a stand-alone tester and not require the eStream client+server programs.	[illegible]
1.11	The appInstallBlock should have support for indicating upgrades at the support site	10
1.12	In the process of creating an eStream set it should be possible for the user to delete file entries and	10

	registry entries manually to "trim" the eStream set if she so desires assuming the user knows what she is doing.	
1.13	The Builder should be run in a clean machine with as few software installed/upgraded as possible.	10
1.14	The Builder should support individual applications in a suite even if the installer of the suite doesn't allow installation of individual applications.	10
1.15	The Builder must be able to create an initial set of cache contents for the eStream client and allow the initial size to be selectable by the user or automatically.	10

5.0 Client Use Cases

5.1 USE CASE: Installation of eStream client code

- Obtain eStream client code bits.
- Install z: file system hooks & setup to have z: mounted at appropriate time.
- Install eStream client code, which services z: file sys requests from local cache or from servers & which handles sideband communication w/ servers, and setup to activate estream client code at time desired by user (boot, login, on demand).
- Install NoCluster.sys to disable page fault clustering at system boot.

5.2 USE CASE: Installation of application

- Obtain AppID & App Server name for installation from ~~DRM~~SLM Server.
- Download AppInstallBlock information.
- Perform initial installation & setup for app, after checking system for previously installed version of app & issuing any appropriate warnings.

5.3 USE CASE: Uninstallation of application

- Remove all registry/DLL/filesys changes associated with app installation.
- Remove all other data associated with application.

5.4 USE CASE: Uninstallation of eStream client code

- Remove z: file system hooks, eStream client code, & nocluster.sys.

5.5 USE CASE: Execution of eStream client code

- Respond to z: file sys requests and detect when new eStream app is referenced.
- Support Client UI requests.

5.6 USE CASE: Execution of application

- Obtain Access Token & list of App Servers from ~~DRM~~SLM Server.
- Contact App Server(s) as desired to obtain file system data.
- Respond to running application's requests, collect usage data. Cache portions of application, file system info, & user preference info.
- Detect server connection issues (apparent loss of connection or connection response below acceptable threshold) & licensing issues; negotiate with A~~DRM~~SLM Server as needed.

6.0 Server Use Cases

6.1 USE CASE: Create an Account

- Customer brings up browser and connects to ASP Web Server
- Screen display shows "create account", customer selects and enters required account info (billing info, owner userid, pword, etc)
- ASP Web server writes account info to Acct DB using **AddAccount** where a unique Account ID is assigned
- Account ID is returned via web page

- Customer brings up browser and connects to ASP Web Server
- Customer enters their userid and pword
- ASP Web server contacts Acct DB, using **AddUser** userid and initial password, gets Acct info and displays to Customer

- Customer selects "add user" and enters required user info (username, address, email etc)
- ASP Web server writes user info to Acct DB updating account info

6.3 USE CASE: Modify Account

(includes disabling an account or user, removing users from accounts, changing pwords etc)

- Customer brings up browser and connects to ASP Web Server
- Customer enters their userid and pword
- ASP Web server contacts Acct DB, passes along userid and pword, gets Acct info and displays to Customer
- Customer selects "update info" and enters desired changes
- ASP Web server writes updated account info to Acct DB

6.4 USE CASE: AddSubscription

- Connect to ASP web server
- Enter account number, username, password
- Verify that user is account admin using **GetUserPermissions**
- Get list of possible subscriptions (using **ListPossibleSubscriptions)**
- Get list of current subscriptions for account (using **ListCurrentSubscriptions)**
- Display in page – User chooses a subscription and license type
- Display a screen to allow the user to configure the license. For a floating license, allow selection of users, etc.
- Call **CreateSubscription** to compose the new subscription for each user and create licenses.

6.5 USE CASE: Building an eStream set:

- Start w/app **CD-ROM**, and a freshly installed OS (plus latest service pack?).
- Install app into Z: drive (could just be a regular network drive)
- A **special system monitor** logs all registry changes and file system changes during the install.
- File system changes to C: during install probably need to be spoofed (or have a registry entry point to Z: instead), especially newly added directories, so need to do the appropriate thing.

- From this log and the actual files as installed on the machine, the **eStream set builder** creates the eStream set, which is a small set of related files.
- Separately, we need to actually set up the app for eStreaming, then run it and collect profile data to seed the initial page prediction map.

The **AppServer UI** (interface to user to control an Application Server on a particular machine) presents the following management functions:

A. Starting a server:

- **AppServer UI** always indicates whether an AppServer process is up and running (and alive w/status), and if present prompts for restarting the current server process.
- Otherwise it goes ahead and starts up the AppServer process and reports any errors.

B. Stopping a server:

- Simple, just stops any running servers, gracefully, perhaps prompting user for ungraceful shutdown if not successful.

C. Install eStream set:

- Each server is configured with a specific eStream set directory, under which it places (in their own individual directories) the actual eStream set contents (a few files on the native file system).
- User indicates to **AppServer UI** where to find the eStream set package provided by Omnishift. **AppServer UI** authenticates the package, and verifies its integrity, and if successful, unpacks and places the constituent files in the server's eStream set directory.
- Note that it is possible for the eStream set directory to live on a file server shared by other **Application Server** machines, so installation may be required only once (otherwise it must happen once per machine, and a separate **AppFarm Manager** is responsible for replicating eStream sets across a farm to ensure the farm machines are symmetric).
- How does the server know a new eStream set is available? Each set is assigned a VolumeID, and the set contents can be placed under a directory with the same [redacted] lists the valid VolumeIDs, which the **Application Server** only reads, so an entry is added at the end of the install procedure. The **AppServer UI** then must send some kind of message/signal to the Application Server to have it resync with the file (and start serving the new app).
- Also note that eStream set install is doable without bringing down the server (or any server in the farm.

- Having done this, it will probably be necessary to notify the ~~DRM~~SLM and **Account Servers** that a new app is available. With some scripts provided by omnishift, this sould be done by a human administrator. They need to know the VolumeID of the app that was installed along with the full name, so that the client can initiate an app install procedure via the VolumeID (the server can then provide the **AppInstallBlock** which probably has a fixed reserved global FileID).
- Questions: What if app is already installed (want to allow reinstall or force remove first)? What if app is being upgraded (probably also should be a remove and then install)?

D. Remove eStream set:

- First we probably have to disable the application on the ~~DRM~~SLM/**Account servers.**
- This probably will require sending some kind of message to the Application Server (if running) to stop serving the given eStream set, and then waiting for any active connections to expire.
- Then we can just remove the entry in the AppList file, and delete the file system image.

E. Configure Application Server:

- The **AppServer UI** presents various configuration options to the user (stuff like logging, port #, threads, etc.) Some may require restarting the **Application Server** to take effect, others may take effect immediately.

Another activity that occurs, automatically, is the processing of profile data. It is not clear what the page prediction map looks like, but clients will periodically send profile data to the **Application Server**, which aggregates it, and must store it persistently, and to allow new clients to benefit from improved prediction. There may need to be a special module that can take the aggregated profile data to modify the prediction map.

6.6 USE CASE: Acquire Access Token

Where are we?

- Client PC has installed the subscribed app and has received a subscription token, and the name/IP of ~~DRM~~SLM Server.
- Customer is accessing an app file and doesn't have an access token for it, yet. (i.e. double clicking z:\word.exe).

Players involved: client – cache mgr, ~~DRM~~SLM Server and indirectly, User/Account/Sub/Rights DB.

What happens:

- Client contacts the ~~DRM~~SLM server and gives: subscription token, user/passwd.
- ~~DRM~~SLM server looks into the user/acc/sub/right DB to
  - Authenticates user and password; may return: "invalid user".
  - Authenticates subscription token; may return: "invalid subscription token"
  - Look at the Accounts container and see if any licenses are available. If so, check it out by creating a new access token and updating the accounts container. It may return: "can't get license".
- Return an access token to the client and a list of app servers.

6.7 USE CASE: Process File Request – steady state

Where are we?

- Client has installed the app and has a list of app servers,
- Client is holding a valid access token that it acquired from the ~~DRM~~SLM server.
- Client, while processing an IRP, needs to access portion of file on the app server.

Players involved: client – cache mgr, app server. NO ~~DRM~~SLM server or no user/account/dub/rights DB.

What happens:

- Client contacts one of the app servers and gives: access token, App ID, File ID, length and file offset.
- App server quickly verifies the expiration date on the access token.
  - It **must not** need to contact the user/account/sub/rights DB to do this. It only cares about the time-validity of the token. If token has expired, return some kind of an error back to the client.
- App server locates the data and sends it back to the client.

NOTE: we are simplifying this quite a bit when discussing the scenarios because [redacted] key question is MUST all app servers host all estream apps ?

6.8 USE CASE: Renew an Access Token – steady state

Where are we?

- Client acquired an access token from the ~~DRM~~SLM server.
- While running the app, client sees the needs to renew the access token. This may happen synchronously when the user touches one of the app files, or by a timer-driven client daemon that periodically renews an access tokens before it expires.

Players involved: client – license manager, ~~DRM~~SLM manager, and indirectly, user/accounts/sub/right/ DB.

What happens:

- Client sends an access token to the ~~DRM~~SLM server.
- Check the time-validity of the access token.
  Assumption: ~~DRM~~SLM server assumes that only valid access tokens can be renewed. An expired token implies a lack of renewal, which implies releasing the license. ~~DRM~~SLM server can try to acquire the license, but there is no guarantee that it will succeed.
  - If token is expired app, goto Scenario: Acquire Access Token.
- ~~DRM~~SLM server accesses the user/account/sub/rights DB to:
  - Generate a new token that will expire some time in the future (configurable parameter).
  - Update the account container in user/account/sub/rights DB.
  - ~~e~~Return the new access token.

6.9 USE CASE: Validate user request for access to an application server

Procedure:
Receive username, password, machineNodeID, subID and appID from the Client
Query AccountDB for license to access application appID in subscription subID
If (no valid license) then
  Send FailureReason to Client
Else
  Send accessToken, appServers to Client

6.10 USE CASE: Add subscribable application from an account

Interface Required:
~~DRM~~SLMServer::AddSubscribedApp(accountID, subID)
Procedure:
Receive accountID, and subID from the Client
Check for valid accountID, and subID on AccountDB
If (no valid accountID or subID) then
  Send FailureReason to Client
Else if (subID is not already subscribed under accountID)
  Add Subscription subID to Account accountID in AccountDB
Send Success to Client

6.11 USE CASE: Remove subscribable application from an account

Interface Required:
~~DRM~~SLMServer::RemoveSubscribedApp(accountID, subID)
Procedure:
Receive accountID, and subID from the Client
Check for valid accountID, and subID on AccountDB
If (exist subID in accountID) then
  Remove Subscription subID from Account accountID in AccountDB
  Send Success to Client
Endif
Send FailureReason to Client

6.12 USE CASE: Monitor/management tools

Interface Required:
~~DRM~~SLMServer::GetTrafficHistory()
~~DRM~~SLMServer::GetUsageInfo(userID, appID, subID, accountID)
~~DRM~~SLMServer::GetCurrentTraffic()
~~DRM~~SLMServer::AddServer(serverID)
~~DRM~~SLMServer::RemoveClient(userID, serverID)
~~DRM~~SLMServer::GetErrors()
~~DRM~~SLMServer::DumpErrors(filename)
~~DRM~~SLMServer::DeleteErrors()
AppServer::GetTrafficHistory()
AppServer::GetCurrentTraffic()
AppServer::GetErrors()

Procedure:
~~DRM~~SLM Servers keep track of traffic info. The monitor/management tool can query the ~~DRM~~SLM/App Servers anywhere for traffic info. Some examples of traffic data:
- Traffic history of particular server on number of clients served per unit time
- Monitor length time a userID used application appID under subscription subID and charged to accountID
- Monitor current load information on all servers (~~DRM~~SLM server and app server)
- Allow admin manually add/remove some servers from the pool.
- Allow admin to kick some clients off the server.

The monitor/management tool can also be used to display a list of errors logged by the servers.
- Monitor errors and be able to categorize by error type
- Monitor errors occurring between certain time periods
- Monitor errors reported by a particular server
- Manage errors to dump the errors to a file
- Manage errors and delete a subset of errors

Finally, the monitor/management tool can check for any illegal accesses.
- Monitor failed attempts to access ~~DRM~~SLM Server with bad password, especially on repeated failed attempts in a short time frame.
- Monitor any attempts to use a particular license and failed.
- Monitor access to ~~DRM~~SLM Server from non-typical IP addresses for a particular account. The server is required to save the history of IP addresses of accesses to a particular subscription account.

6.13 USE CASE: Adding a new application server.

Summary:
An application server's functionality is to provide applications eStream sets to client application. An application server is generally added to the system to provide greater scalability and/or to provide additional application support.

Actors:
1. ASP administrator: Responsible for installing the server and the applications
2. ~~ADRM~~SLM Server(s): The ~~ADRM~~SLM server needs to be notified of the presence of an additional application server and the services it provides.

Inputs:
1. Application server(AS) installer
2. Application eStream sets. These may be available from one of the following location: AS installer, some other AS or Farm Manager Server(some central repository) .

3. ~~DRM~~SLM Server location(This input may not be required based on scalability solution that we decide on).

Processing:

1. Using the AS installer install the application server.
   a. <Server install use case to be added here? Later>
2. Copy the Application eStream sets. There are several options here:
   a. Provide the eStream sets as a part of the installer.
   b. Provide a script to ftp to another Application server and copy the eStream sets.
   c. Provide a management tool to manage the copying of the eStream sets. From the ASP's perspective this is the best solution. A tool which provides tracks the application would be useful to manage the load.
3. Configure the server. The server needs to know the additional application sets that it supports(? This may not be required).
4. Start the server.
5. Register the server with other ~~DRM~~SLM servers. The following options apply:
   a. Multi-cast the "new server and services" message to the ~~DRM~~SLM servers.
   b. Register the server to a local object server which in turn notifies the object servers across the system. CORBA model supports this.
   c. Using the resonate model(described below), all appservers are essentially the same server. ie Address app.foo.com will point to a set of app servers. A new server enabled will resonate software will automatically register itself with the resonate scheduler. (How do we make the resonate scheduler aware of the applications available on the app servers?)

Outputs:

1. The App server is installed and running with a set of applications available on it.

6.14 USE CASE: Removing an Application Server.

Summary:

An ASP administrator may decide to remove an application server from the system for various reasons. Removal of server from the system would result in notification to the rest of the ~~DRM~~SLM servers that it is no longer available for servicing the objects.

Actors:

1. ASP administrator.
2. ~~DRM~~SLM Servers.

Inputs:

1. Application server running on the machine.
2. A~~DRM~~SLM Server(s): The A~~DRM~~SLM server needs to be notified of the presence of an additional application server and the services it provides.

Processing:

1. Stop the application server. This will result in the Application server informing the rest of the ADRMSLM servers that it will no longer take any requests. This in turn may result in an application being unavailable for usage. Depending in the framework used, this can be done in one of the following ways:
    a. Multi-cast the message to the ADRMSLM servers.
    b. Just stop the server in the CORBA framework. The local ORB server will notify the unavailability of the resource to the rest of the framework.
    c. Using the resonate model to scale would imply that you just stop the server the resonate agent on the server will notify the resonate scheduler to deregister the servers. (However its not clear if you can also deregister the objects served by the server.).

Outputs:

1. ADRMSLM servers are notified of the removal of the resource.

6.15 USE CASE: Add a new ADRMSLM server.

Summary:

The ASP provider may decide to add an additional ADRMSLM server to enhance the performance of the system. The additional ADRMSLM server added to the system should be accessible to the ASP's Web Server so that it can direct the clients to the DRMSLM server. (This may not be required if we deploy the Resonate model of scaling).

Actors:

1. The ASP administrator.
2. The ASP Web server.

Inputs:

1. ADRMSLM installer
2. Web Server location(This input may not be required based on scalability solution that we decide on).

Processing:

1. Using the ADRMSLM installer install the application server.
   <Server install use case to be added here? Later>
2. Start the server.
3. Register the server with ASP Web Servers. The following options apply:
    a. Multi-cast the "new server and services" message to the Web servers.
    b. Register the server to a local object server which in turn notifies the object servers across the system. CORBA model supports this.
    c. Using the resonate model(described below), all ADRMSLM are essentially the same server. ie Address adrmslm.foo.com will point to a set of ADRMSLM servers. A new server enabled will resonate software will automatically register itself with the resonate scheduler.

Outputs:

The ADRMSLM server up and running.

7.0 Builder Use Cases

7.1 USE CASE: Install Monitoring

- Query builder for CD media and installation executable(s)
- Monitor various registry and file updated during installation
- Merge installation data for all applications in a suite
- Relocate files from C: to Z: directory
- Create appInstallBlock and package the appInstallBlock with the application files

7.2 USE CASE: Profiling

- Query builder for application executable(s)
- Monitor sequences of file accesses from OS to the file system as profile data
- Identify the subset of the profile data as the initial cache contents
- Merge profile data and initial cache contents into the corresponding appInstallBlock

8.0 Key Applications for eStream 1.0

Winstone99:
Business:
Corel®WordPerfectSuite8: Quattro®Pro 8, WordPerfect®8, Netscape Navigator®4.04
Lotus® SmartSuite®: Lotus® 1-2-3® 97, Word Pro® 97, Netscape Navigator® 4.04
Microsoft®Office97: Access97, Excel97, PowerPoint®97, Word97, NetscpNav® 4.04
High-end:
Adobe® Photoshop® 4.01, Adobe® Premiere® 4.2, AVS/Express® 3.4,
Bentley System's MicroStation® SE, PV-Wave® 6.1, Microsoft® FrontPage® 98,
Microsoft® Visual C++® 5.0, and Sonic Foundry® Sound Forge® 4.0.
Content Creation Winstone 2000:
Adobe Photoshop 5.0, Adobe Premiere 5.1, Macromedia Director 7.0,
MacromediaDreamweaver2.0, NetscapeNavigator 4.6, Sonic Foundry Sound Forge 4.5

Please note that release of Business Winstone 2000, which was originally slated for [redacted], has now been postponed until the Fall Comdex & will be called Winstone [redacted]. As soon as the contents of this suite are released, we should move quickly to assess our support for its application set.

eStream 1.0 Server Scaling Estimate

Anne Holler * [redacted] * *Version 1.0*

Introduction

This document presents an estimate of server scaling for the eStream 1.0 product as compared with its chief competitor, the Citrix product as deployed by Personable. The document presents relevant attributes of the basic application execution model for each of the two products, discusses and gauges the impact of the areas in which server scaling differs between them, considers the effects of additional attributes of the two products on server scaling, & finally summarizes the differences in expected server scaling in terms of a number. Please feel free to challenge the assumptions, methodology, & calculations herein, now & as we move forward through the design & implementation phases.

In the process of developing this server scaling estimate, certain assumptions about user, system, & program behavior are made, & certain design/implementation goals of the eStream 1.0 product are assumed. This material is listed in separate sections at the end of the document for ease of reference.

This work does not intend to imply that the user experience of the eStream 1.0 & Personable/Citrix products is expected to be comparable with respect to the relative server scaling point identified. Interactive response differs between the two products; first-hand experience with server-based applications running on New Moon & Personable/Citrix and client-based applications running on the eStream prototype suggests that the former are sluggish on an ongoing basis with respect to activities such as selecting from pull-down menus & the latter are as responsive as native wrt such activities, with noticeable delays engendered only when heretofore unused portions of the application's functionality are exercised. Reliability also differs between the two products; smooth fail-over of an active Personable/Citrix application to another server is not supported, whereas such fail-over is included in the eStream 1.0 design.

This document does not address an area in addition to server scaling that may be of competitive interest to ASPs; that area is network bandwidth differences between eStream 1.0 & Personable/Citrix. Though it might seem intuitive that sending application pages across a network consumes more bandwidth than sending user input & display output, aggressive client caching ameliorates the traffic associated with application paging, whereas the traffic associated with the display output can be quite substantial, according to Harwood's book "WNT Terminal Server & Citrix Metaframe" [HARW99]. It may be worthwhile to collect and compare bandwidth data wrt the two products.

Acknowledgements

Amit Patel provided key insights. Any mistakes/bogosity are mine.

Application Execution Models

The following are basic attributes, with respect to application server scaling, of the application execution models of Personable/Citrix and eStream 1.0:

Personable/Citrix Client/Server Application Execution Model
Application executes on server
On app page faults & app data reads, page loaded from server disk
Server handles incoming network traffic for all keyboard & mouse events
Server handles outgoing network traffic for bitmap display update

eStream 1.0 Client/Server Application Execution Model
Application executes on client
On app page faults & app data reads, page loaded from client cache, excpt miss to server
Server handles incoming network traffic for client cache misses
Server handles outgoing network traffic for client cache misses

Application Server Scaling Comparison

The impact on server scaling of executing applications on the client, rather than on the server, is expected to be large. Processor & main memory overhead are associated with executing applications, including the overhead for fielding interrupts such as mouse & keyboard events. According to HARW99, processing power for running applications is typically the biggest Citrix product bottleneck, and that is reinforced by the variance in Citrix scaling numbers reported wrt Personable [Ernie] and wrt another ASP [Amit], for which the main differences seem to be application execution overhead. HARW99 indicates that Citrix scales at 10 to 45 applications per processor, largely due to execution overhead (though some overhead is due to processing page faults & accessing application data, which is considered in the next paragraph). It is somewhat difficult to understand how to model the relative benefit for this difference (in the sense that an *infinite* number of applications can run on a processor that they are not actually using to execute!); it seems conservative to assume we get at least as much benefit from this factor as we do from reducing cache miss overhead on the server, so let us have this factor double whatever benefit we project from the factor considered in the next paragraph.

The impact on server scaling of processing application page faults & application data reads on the client in most cases, rather than on the server, is expected to be measurable. For the purposes of the server scaling estimate presented in this document, let us assume that the server overhead to fetch a page from disk, whether the request came from server execution or from client request, is comparable. (Although we can construct file server technology in which client requests take less time than server requests, let us assume that the overhead to encrypt the response consumes that time savings). Also, let us assume that the same number of page faults occur on the client & on the server (which server partitioning for Personable/Citrix could render untrue.) Hence, the difference in server

overhead for application file accesses is estimated to be equivalent in scale to the reduction in the number of references, which is derived from the client cache miss rate. Assuming a client cache miss rate of 2%, each eStream application server can handle 50 times as many clients with respect to this attribute of server overhead. Doubling that amount due to the factor described in the previous paragraph, we estimate that each eStream application server can handle 100 times as many clients as each Personable/Citrix application server.

It is somewhat difficult to know how to compare the network interface overhead component of server scaling between eStream 1.0 and Personable/Citrix. The loading constraints associated with executing applications on the server are expected to limit the amount of network overhead presented to a Personable/Citrix application server. Depending on the kind of application, significant traffic is generated to support client displays, but HARW99 identifies network bandwidth overhead [discussed in the Introduction section] – not server overhead - as the scaling problem engendered by this traffic. Each eStream 1.0 client generates little server network overhead, but the reduction in server load due to client application execution allows more clients to be connected to a given server, possibly straining the network-oriented portions of an eStream application server. At this point, let us assume that server network interface overhead does not materially impact the relative server scaling of the two products.

SSL encryption can dramatically decrease server scaling; by more than 70%, according to Igor Balabine. He indicates that third party SSL accelerators should be employed to remove this overhead.

Additional Server Scaling Considerations

For eStream, application installation induces load on the application server to deliver to the client the contents of the AppInstallBlock, which may contain registry & file spoofing information, initial cache & profile data, file system structure information, etc. Personable/Citrix does not have a comparable feature. Installation is expected to be an infrequent occurrence and Omnishift is expected to suggest/provide mechanisms to smooth out (or specially handle) high peak demand at particular points in time, including product or application launch/upgrade. Let us assume that this (managed) overhead reduces application server scaling by the equivalent of 0.5% cache miss. Adjusting the running total by this amount implies that an eStream application server can handle 67 times as many clients as a Personable/Citrix application server can.

[redacted]

does not have a counterpart on Personable/Citrix [except for page fault clustering ;-) !]. Given that eStream 1.0 is targeted at fat clients, client prefetching (which is redundant wrt a comparably warmed client cache) is expected to be used sparingly. Let us assume that prefetching adds the equivalent of an additional 0.5% cache miss rate; the intuition here is that prefetching is essentially engendered when cache misses occur (i.e., when we are exercising parts of the app we have not exercised before) and that we get unneeded pages a measurable percentage of the time. Updating our running total by this amount

means that an eStream application server can handle 50 times as many clients as a Personable/Citrix application server can.

Both eStream & Personable/Citrix products include several logical servers in addition to the application server. Both have an ASP Web Server portal to an ASP's account services, from which a user can obtain billing information, get a list of available applications, subscribe to new applications, etc. For both, the ASP web server interfaces in some way with an account database of presumably comparable complexity. Both eStream & Personable/Citrix have server functionality involving getting the license to run an application, which is expected to cause similar database overhead; in eStream 1.0, this process involves getting an AccessToken from an ADRM server. However, Personable/Citrix does not have eStream 1.0's concept of renewing an AccessToken. It is expected that the eStream 1.0 design will take special care that renewal does not add significant overhead to the eStream 1.0 ADRM server (by specifying nontrivial billing granularity & AccessToken renewal frequency, by ensuring renewal is lightweight, perhaps by having some explicit mechanism aside from AccessToken renewal for token cancellation in the event of client failure/disconnect, etc.). eStream 1.0 has the concept of records containing application profile information being uploaded to a server; it is not known that Personable/Citrix has any comparable feature (although the product likely has much other user information recorded at the server, including preferences, etc). It is expected that eStream 1.0 design will emphasize minimal server impact for handling this data. [Profile data may not be a core deliverable for competing with Personable/Citrix.] In summary, it is assumed that server scaling for auxiliary servers is comparable between eStream 1.0 & Personable/Citrix.

Conclusion

Based on the discussions in the previous sections, eStream Server Scaling expected to be significantly higher than that of Personable/Citrix. Considering client execution benefits, client caching benefits, application installation overhead, and prefetching overhead, eStream server scaling is expected to be approximately 67 times higher than the Personable/Citrix competitive product. For some given Personable/Citrix application server that can handle 20 clients, an eStream application server can handle 1340.

Assumptions Underlying Server Scaling Estimate

User's application usage pattern [what is run, for how long, what features] does not change materially depending on whether s/he is using eStream 1.0 or Personable/Citrix.

[illegible] estimates for his network bandwidth evaluation, but it might be interesting to see if our in-house use of common applications matches those estimates.

Overall application server capacity is adequate for both products. Personable/Citrix may deny application server access to clients when insufficient overall application server capacity is available, whereas eStream 1.0 may continue to allow clients to access the

servers without some explicit client capacity cutoff point, given the usual small impact of each additional eStream client. However, it is possible that an extreme performance collapse could occur on eStream, if client load were to grow so large compared with the capacity of eStream's application servers that the lack of server response caused an escalating number of redundant requests from eStream clients retrials. This situation needs to be avoided, in the eStream 1.0 design and/or in its deployment.

The increased client servicing capacity afforded an eStream 1.0 application server will not uncover some insurmountable system bottleneck never encountered with the limited client servicing capacity possible on the Personable/Citrix server, including areas such as maximum number of threads, processes, sockets, buffer size for socket send or receive, socket listen queue length, network buffer cache, maximum number of file handles, etc.

eStream 1.0 clients are fat enough to allow adequate client caching to hit or better the client cache miss goal of 2%. Thin clients (which are not the intended design target for eStream 1.0) or fat clients with inadequately sized client caches would increase server overhead for paging requests & network traffic beyond the levels estimated in this document.

Design Goals Supporting Server Scaling Estimate

Overall client cache miss rate is less than 2%. Reaching the eStream 1.0 client performance goals also reinforces the need for a low client cache miss rate. We have not collected data indicating how large a client cache would be needed to hold application pages and file metadata associated with typical application usage as represented by the Ziff-Davis benchmark runs. I think it would be useful to have such data, since it may influence client cache design & effective cache management policies.

AppInstallBlock server overhead is no more than the equivalent of an extra 0.5% cache miss rate. Installation is expected to be relatively rare, & downloaded material is expected to be kept at the minimum size necessary to reach our functionality & client performance goals.

Client wasted prefetch overhead is no more than the equivalent of an extra 0.5% cache miss rate. With fat clients & large warm caches, prefetching is expected to be kept at a minimum for eStream 1.0; again, data gathering related to this area may be useful.

eStream License Manager (LSM) Low Level Design

Charles T. Booher
[redacted]

Functionality

The LSM tracks the users subscriptions to ASP accounts. It is part of the client component and runs entirely in user space. The LSM is part of the client UI program that also contains the Cache Manager (ECM), Application Install Manager (ECM) and client user interface components. The LSM determines when an application has a right to run on a client machine. This is done through the acquisition and management of access tokens. The LSM also provides a server list to the ECM (Cache Manager) for each application that is subscribed. The LSM has two main tasks.

Manage and provide lists of servers for the ECM

Provide access tokens to the ECM and renew them before they expire.

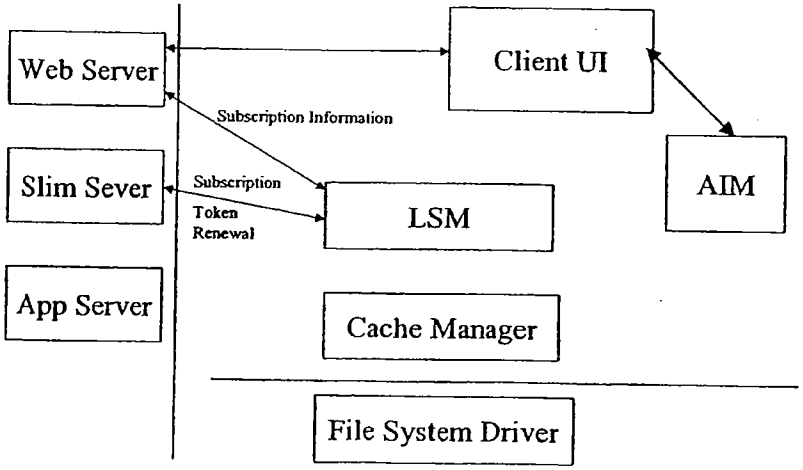


Figure 1 LSM data flow

What does the LSM do?

The LSM manages a set of tables inside of an mdb database. This database can be accessed with ADO or ODBC. This database includes the following tables.

Application Table: This is a list of all the applications subscribed.

User Table: List of User account information

Application Server Table: This is a list of application servers for each application subscribed by the client.

Slim Server Table: This is list of all Slim Servers where access tokens can be requested for renewal by an application.

Access Token Table: This is a list of all access tokens granted by the SLIM server or installed with an application subscription.

When does the LSM operate?

An Application is Subscribed

When an application is subscribed, the Application Table, Application Server Table, and Slim table must be updated to reflect the new subscription. An expired token is added to the Access token database. The normal token renewal processes with update the token on the first installation, but the sleep loop for the token thread is interrupted so that the token renewal thread is woken up. The token renewal thread will see the expired token for the newly subscribed application and issue a renewal request to the SLIM Server.

Application subscription tasks:

1. Update the tables
2. Wake up the Access token Renewal thread.

A user Logs on

When a user logs on to the system the client will get a complete list of Application Identifiers numbers from the web server and compare them with the application table. If there are any applications that are no longer subscribed or new application the system does not know about a Message box will appear asking the user if they would like to update their subscription information. The synchronization of application sets from client to client is only handled by the client user interface. Another task started at logon time is the Token renewal thread. The token renewal thread scans the token list to see if any token is ready to be renewed. The token renewal thread is sleeps on a long timer and is only woken up if a new access token is added to the token database. When the sleep timer on the thread

expires, the thread will scan the list of access tokens to see if any of them have expired or are near expiration.

Logon Tasks:

1. Establish a logon with the ASP Web server. Get a list of all applications ID numbers and check if any have been added or deleted. If there are any adds or deletes force the user to go to the Client administration UI.
2. Update the Application Server Table
3. Update the Slim Server Table
4. Start the token renewal thread.

An application is started

When an application is started, the ECM will request an access token from the LSM. If the token is expired then a Message box will be presented to the user with information about the expiration of the application. The user can go from that User Interface to the client UI that administrates application subscriptions.

An Application is un-subscribed

When an application is un-subscribed, the corresponding records in the Application Table, Application Server Table, Slim table, and Access token table must be deleted.

Data type definitions

These tables are stored in an mdb database file. This database file can be accessed using ODBC or ADO database drivers.

Application Table

A master table of all application that are subscribed.

Name	Application ID	Application name	RootFileNumber	ForcedUpgrade	Message
Type	GUID 128 bit	String	128 bits	8 bits	String

This table describes the data in the eStreamAppUpgradeInfo Structure[1]. This table will link application names with Application ID numbers.

Application Server Table

This table will link Application Servers with subscribed Applications. This data is used by the ECM to get file information.

Name	Application ID	Application Server IP
Type	GUID 128 bit	IP Address 32 bit

A particular client application can have more that one server.

Slim Server Table

This table is a list of all servers where access tokens can be requested. An application can have more than one Slim Server.

Name	Application ID	Slim Server IP
Type	GUID 128 bit	IP Address 32 bit

[1] Software License and Management Server Amil Patel, Bhaven Avalandi, Michael Bechman, Sameer Panwar

Access Token Table

Name	ATID	UserID	AppID	Expiration Date
Type	GUID	GUID	GUID	Date

Access Tokens are requested when an application is first subscribed. They are renewed before they expire.

User Table

Name	User ID	User Name	Password Hash
Type	GUID	String	String

A users password is never saved. The user password is fed to a one way hash function and the hash output is what is saved to the database.

Interface definitions

Function 1

```
BOOL GetToken{
   GUID AppID,
   int size,
   eStreamAccessToken * TokenData);
```

Input:

GUID AppID: the Application ID for which a token is being requested.

Int size: the size of the token data structure

eStreamAccessToken * TokenData: the token structure.

Output:

The function will return True if a valid access token has been granted, false otherwise. The function will handle client UI for expired tokens.

Comments:

The eStreamAccess Token is defined in the slim server documentation.

Errors:

Application ID not found.

```
DWORD GetAppServer(GUID appID, int servernum)

Input:
   GUID AppID, this is the application ID for
   which a server is being requested.

   Int servernum, this number is usually 0, if
   a particular server is not available then
   the ECM will ask for the next server.  If
   the Server table does not have a server then
   this function will return -1

Output:

32 bit IP address

Comments:
A return code of -1 indicates that we have run
out of servers

Errors:

Application ID not found.
```

Component design

Database access

Access to the database tables is through either an ODBC or ADO interface. The AIM can access the databases directly though these interfaces. The program code that updates these tables is considered part of the AIM, but the databases themselves are considered part of the LSM.

Token Renewal Thread

The token renewal thread is a worker thread that is part of the Client executable. This thread is started with AfxBeginThread.

```
UINT MyThreadProc( LPVOID pParam )
{
  CTokenRenewalObject* pObject = (CTokenRenewalObject *)pParam;

  if (pObject == NULL ||
    !pObject->IsKindOf(RUNTIME_CLASS(CTokenRenewalObject)))
  return 1;  // if pObject is not valid

  // do something with 'pObject'

While(1)
{
        if(CheckForTermination())
                return 0;
        pObject->AccessTokenTable->MoveFirst()
        while(!pObject->AccessTokenTable->IsEOF())
        {
                if (NearOrAtExpiration(pObject->AccessTokenTable->Expiredate))
                {
                        pObject->RenewToken(pObject->AccessTokenTable->TokenID);
                }

                pObject->AccessTokenTable->MoveNext();
        }
```

```
        ComputeSleepTime();
        Sleep();
}


  return 0;  // thread completed successfully
}

// inside a different function in the program
.
.
.
pNewObject = new CMyObject;
AfxBeginThread(MyThreadProc, pNewObject);
.
.
```

Logon Initialization

This block of code is part of the Client executable when the client first starts up.

```
void ClientStartup(void)
{
   LogontoASPWeb();
   GetSlimServerTable();
   GetAppServerTable();

   StartTokenRenewalThread();

}
```

Testing design

The LSM is part of the Client Executable. This test plan will become part of the client executable test plan.

Unit testing plans

1. Initial Application Subscription Install. Does the AIM update the LSM tables correctly? This can be checked using Access.
2. Contact the ASP web server and Update the Application Server Table.
3. Contact the ASP web server and get a list of applications.
4. Does the Client UI respond when the list of applications is changed?
5. Force tokens to expire using Access and see if they are renewed by the Slim Server
6. Force access tokens to expire and prevent the Slim server from granting them.

Stress testing plans

Stress happens when Application Server and Slim Servers go down. Stress testing can be accomplished by running subscribed applications and then shutting down servers.

Coverage testing plans

Every application that eStream is capable of serving will need to be tested.

Cross-component testing plans

The LSM communicates with

ECM

AIM

Web Server

Slim Server

Client UI Components.

Upgrading/Supportability/Deployment design

This component is part of the client executable program. When the client executable program is upgraded then this component will be upgraded.

Open Issues

Every time the user logs on to the client a list of currently subscribed, applications must be downloaded from the server to insure client synchronization. Right now we don't have this function defined on the server side.

Omnishift C/C++ Coding Standard

[redacted]

Omnishift Confidential

The following proposal is based on the C++ coding standards document available at http://www.possibility.com/Cpp/CppCodingStandard.html. This document will concisely present the coding standards from the coding standard document. The reader should refer to the original document (linked above) for a detailed explanation of the standards. This document has the following sections:

NAMES: This section contains the naming schemes.
ERRORS AND ERROR CODES: This section contains the error formats and the error codes to be used by eStream 1.0 client and server components.
FORMATTING: Code layout and formatting guidelines.
COMMENTS: Guidelines for applying comments to the code.
LOOSE END: Loose ends.

Table of Contents:

NAMES

ID	RULE
Variables	• Use upper case as word separators, lowercase for the rest of the word. • No underbars ('_') • First letter of the variable is uppercase unless it is prepended with some other letters(see below). **Examples:** short Status; unsigned long TimeOfDay;
Pointers	• Prepend the variable with p. **Examples:** string* pName; char** ppValue;
Class Names	• Use upper case letters as word separators, lower case for the rest of a word • First character in a name is upper case • For externally exposed components, use the first 3 words to denote the component. • No underbars ('_') **Examples:** • class ConfigurationManager • class Config
Library Class Names	• Prefix the classname with OT **Examples:** • class OTHttpListener • class OTServer
Class Method Names	Same rule as for class names except for interfaces where the rule is: • Prefix the interface with the component's name. **Examples:** • ECMGetFileId • MonitorGetServerSet
Class Attribute Names	• Attribute names should be prepended with the character 'm'. • After the 'm' use the same rules as for class names. • 'm' always precedes other name modifiers like 'p' for pointer. **Examples:** int miLen; char* mpValue; string* mpValue;
Global Variables	• Global variables should be prepended with "g". Examples: int gFlag; Logger gLog; Logger* gpLog;

Global Constants	• Global constants should be all caps with '_' separators. Examples: `const int A_GLOBAL_CONSTANT= 5;`

Type Names	• When possible for types based on native types make a typedef.
	• Typedef names should use the same naming policy as for a class with the word *Type* appended.
	Examples:
	`typedef uint16 ModuleType;`
	`typedef uint32 SystemType;`

#defines and Macros	• Put #defines and macros in all upper using '_' separators
	Examples:
	`#define MAX(a,b) blah`
	`#define IS_ERR(err) blah`

Function Names	• Use upper case letters as word separators, lower case for the rest of a word • First character in a name is upper case • No underbars ('_') **Examples:** • int SomeBloodyFunction()

Enum Names	• all upper using '_' separators
	Examples:
	`enum PinStateType`
	`{`
	`PIN_OFF,`
	`PIN_ON`
	`}`

File Names	• File should be all lower case • File name format should be <component>_<sub-component>.* **Examples:** monitor_heartbeat.cpp core_configmanager.c

ERRORS AND ERROR CODES

The following pound defines should be used for returning all successes and failures.

#define SUCCESSWITHINFO -1
#define SUCCESS 0
#define FAILURE >0 (The number representing an Error ID).

All error messages will be prepended with an error code. The format for error code will be as follows:

[ERRORID] [Severity] [Error Message]

where,

1. ERRORID are unique across the system.
2. Severity can be one of the following:
    a. 1-Low : A warning which can be ignored.
    b. 2-Medium: A warning which needs to be looked into.
    c. 3-High: Recoverable error in the component.
    d. 4-Critical: Irrecoverable error. Needs admin assistance.
3. The error message in itself should have the following format:
   [COMPONENT]:[ERROR MESSAGE]:[WORK AROUND]

Error Ids distribution for client and server are as follows:

0-1000 Server Internal Error Codes.
1001 – 8000 Server Error Codes.
8001 – 9000 Client Internal Error Codes.
9001 –16000 Client Error Codes.

FORMATTING

The following formatting policies should be followed by all code.

Braces Policy.

```
if ( 0 == a)
{
...
}
else
{
```

...
}

Indentation/Tabs/Space Policy

Use the standard Visual C++ settings which are(using Tools->Options->Tabs menu):

Indent Size: 4
Auto Indent: Smart
100 previous lines used for context.

White spaces should be spaces and NOT tabs.
VC++: Select "Insert Spaces" option. (This is NOT the default).
Emacs: Refer http://www.delorie.com/gnu/docs/emacs/emacs_205.html

Line Size etc.

1. Line size should not exceed 78 characters.
2. There should be one statement per line. The following piece of code violates this principle.

```
if (a>b) a++;
```

Method/Functions Formats.

1. Methods should preferably be less 50 lines of code.
2. Methods should not have more than 4 levels of nesting.
3. Methods should preferably be re-entrant. Non-reentrant methods should be clearly marked as such.
4. Each method/function should be preceded with a comment describing the method:

```
/*************************************************************
FUNCTION:
INPUTS:
OUTPUTS:
DESCRIPTION:
ERRORS:
*************************************************************/
```

COMMENTS

1. Every decision should have comments. The following keyword are associated with decisions:
    a. if, else
    b. while, continue
    c. switch, case, default, break
    d. goto
    e. return
2. Every class should have comment header with the following format:
    /**
    CLASS NAME:
    DESCRIPTION:
    FRIEND CLASSES:
    INCLUDES:
    LIBRARIES:
    **/
3. Every function/method should have a header. (described above).
4. Every file should have a header describing the contents of the file.
5. Every directory should have a README describing the contents of the directory.
6. Make GOTCHAS explicit. Use the following format for gotchas.

    - **:TODO: topic**
      Means there's more to do here, don't forget.
    - **:BUG: [bugid] topic**
      means there's a Known bug here, explain it and optionally give a bug ID.
    - **:KLUDGE:**
      When you've done something ugly say so and explain how you would do it differently next time if you had more time.
    - **:TRICKY:**
      Tells somebody that the following code is very tricky so don't go changing it without thinking.
    - **:WARNING:**
      Beware of something.
    - **:COMPILER:**
      Sometimes you need to work around a compiler problem. Document it. The
    - **:ATTRIBUTE: value**
      The general form of an attribute embedded in a comment. You can make up your own attributes and they'll be extracted.

LOOSE ENDS

The following section notes some loose ends which do not fall in any of the categories above:

1. Always **initialize all variables** every time.
2. Use **header file guards** against multiple inclusions of the header file. The guards would look like:

   ```
   #ifndef ClassName_h
   #define ClassName_h
   ....
   #endif // ClassName_h
   ```
3. Object constructors should just initialize data. (They cannot return errors). Explicit Initialize() calls should be made to do any involved work.
4. Use **continue** and **goto** sparingly.
5. Be **"const" correct**. Use "const" wherever and whenever applicable.
6. All classes must have a **Default Constructor** and a **Copy Constructor**

THIS PAGE BLANK (USPTO)

QUESTIONS TO ASK AN ASP

ASP Overall

1. What is the current solution deployed by the ASP? Why?
2. What applications are most interesting to the ASP? What platforms do these applications run on?
3. What applications do they currently support? What are the licensing models for these applications? *
4. How often are the applications upgraded? *
5. What are the bottlenecks faced with the current solution?
6. How is user data handled? Is there an issue with the handling of user data?
7. What billing systems does the ASP use? *
8. What pricing models does the ASP employ? What pricing models are important to the ASP for the future? *
9. What is the growth estimate in terms of the number of users?
10. Who are the ASP's key partners?
11. Who are the ASP's key suppliers (or dependencies)?
12. Who are the ASP's key competitors?
13. How does the ASP differentiate its products?
14. Does the ASP have a QOS solution?
15. What are the ASP's major security concerns?
16. What is limiting the ASP's growth?
17. What are the ASP's current bandwidth requirements?
18. How significant an operating cost is this expense?
19. How does the ASP expect this bandwidth requirement to change over time?

ASP Infrastructure

1. What are the platforms on which the ASP applications run?
2. What are the platforms on which the ASP servers run? *
3. Does the ASP maintain the infrastructure? If not, where is it outsourced and why?
4. What are the current data center space needs?
5. How significant an operating cost is this expense?
6. How does the ASP expect this space need to change over time?
7. What kinds of load balancing systems does the ASP deploy? *
8. What are the app-servers (middleware) used by the ASP?
9. What ports do the servers use in the current solution? *
10. How many users does each server adequately support?
11. What are the most important features (wish list) for the app-server infrastructure?
12. Is the ASP willing to add h/w accelerators for encryption?
13. How much unplanned down time has the ASP been experiencing?
14. What have been the main causes of these down times?

ASP end user

1. Is it ok to ask the user to do reboots? *
2. How big of a cache can eStream assume? *
3. Does eStream need to support local installations of the same application? *
4. What is the minimum performance tolerated by the end user?
5. What is the typical bandwidth experienced by the end user?
6. Is the end user always connected?
7. Is there a need to allow for offline access of applications?
8. Can eStream assume that the end user has the ability to install drivers?
9. Is the end user always behind firewalls?
10. What is the accepted level of security for the client-server communication? *
11. What is the accepted level of piracy protection needed on the client side? Does the ASP need its own Intertrust kind of solution? *
12. What is the acceptable delay for installing an application? (Basically, what is the maximum size of the AppInstallBlock acceptable to the ASP?) *
13. Does the ASP need an ability to terminate end users in the middle of a billing cycle? *
14. Can the end user run the same application on multiple clients at the same time? *
15. If the license expires, can the client continue running if the application is in the cache? *
16. Is collecting profile data from the end user acceptable? Is the profile data of use to the ASP?

Deployment

1. What makes for a "good" deployment?
2. Describe a "good" deployment?
3. How would the ASP like to work with a solution provider to deploy their solution in the ASP environment? (i.e. solution provider installs for ASP; help the ASP install; ASP installs alone, other)
4. How would the ASP look to deploy a solution like eStream? (i.e. phased implementation; full-blown roll-out, other)
5. How would the ASP want to receive the eStream solution software? (i.e. physically, electronically)
6. What things would be of concern to the ASP in deploying eStream in the ASP environment?
7. What sort of technical questions would the ASP want answered before considering deploying a solution like eStream?
8. How would the ASP want to receive training on the solution and it's installation and administration?
9. Would there be a need for a Certification Program for eStream Administrators?
10. What sort of tools/abilities would the ASP require to manage and monitor a solution like eStream?
11. What kind of monitoring solutions does the ASP IT team use?

Support

1. What would the ASP like to see/require in an SLA?
2. What makes for a "good" support experience?
3. Who, that the ASP is currently working with, provides "good" support and what happens?
4. What is missing in the current SLA's?
5. What would the ASP like to see/require in support provided by a solution provider?
6. What is missing from current solution provider support offerings?
7. To what extent would the ASP work with a solution provider in remotely troubleshooting problems?
8. Would allowing us remote access into a "Monitor" tool/DB or Log running on the ASP system to effectively troubleshoot a problem be acceptable? How would the ASP see this working?
9. Would the ASP provide first line support to the end users?
10. What does the end user support offering consist of?
11. What would the ASP require from us in order to be able to offer end-user support?
12. Does the ASP actually provide that support or does it have a 3rd party do that on its behalf?
13. How would the ASP like to be notified of upgrades and patches?
14. How would the ASP like to receive upgrades and patches?

Citrix

1. What does the ASP like about MS Terminal Services and Citrix offerings?
2. What does the ASP NOT like about MS Terminal Services and Citrix offerings?
3. What 3rd party tools does the ASP use in addition to or in place of Citrix's offerings and why?
4. How well does the Citrix solution scale? Perform?
5. How would the ASP rate Citrix's support offerings?
6. If the ASP were redesigning Citrix Support what would be done differently?
7. How did the ASP deploy the Citrix solution?
8. What sort of issues did the ASP run into in deploying the Citrix solution?

eStream Server Component Framework
Low Level Design

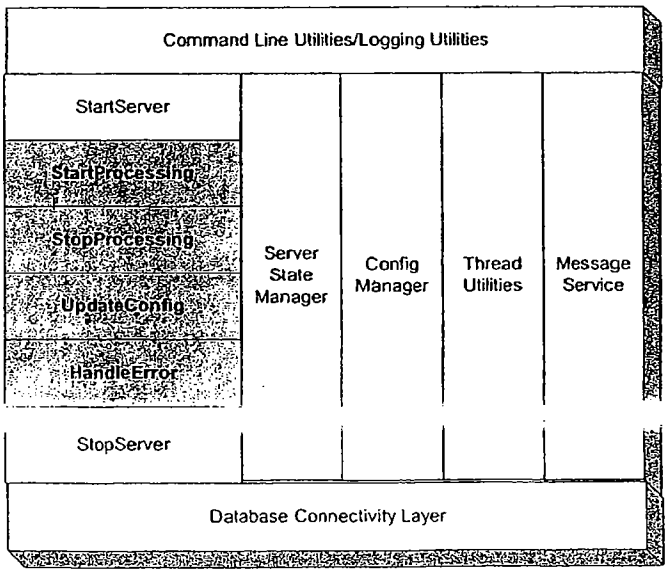
Michael Beckmann
[redacted]

Functionality

The *Server Component Framework* provides a common basis on which server components are implemented. The framework provides a number of services such as common server initialization and configuration, messaging, state management, logging, and error handling. The component framework ties together many of the core utilities provided for the server components.

The advantage of the framework is that heterogeneous server components can be managed in a consistent manner with the expectation that all server components will communicate and behave consistently within the system.

All server components with the exception of the web server will be built on top the *Server Component Framework.* To make use of the *Server Component Framework,* a specialized server component will need to extend the framework by implementing the methods high-lighted in gray. Implementing these interfaces makes the specialized server component "plug-able" within the framework.



The following table give a brief description of each of the routines that need to be specialized by each server component to make it plug-able into the Server Framework:

StartProcessing	Specialized server component routine to request the server component to start processing work.
StopProcessing	Specialized routine to request the server component stop processing work and transition into an idle state
UpdateConfig	Specialized routine to dynamically update configurations while a component is either in the processing or idle state.
HandleError	Specialized routine to handle the occurrence of an error

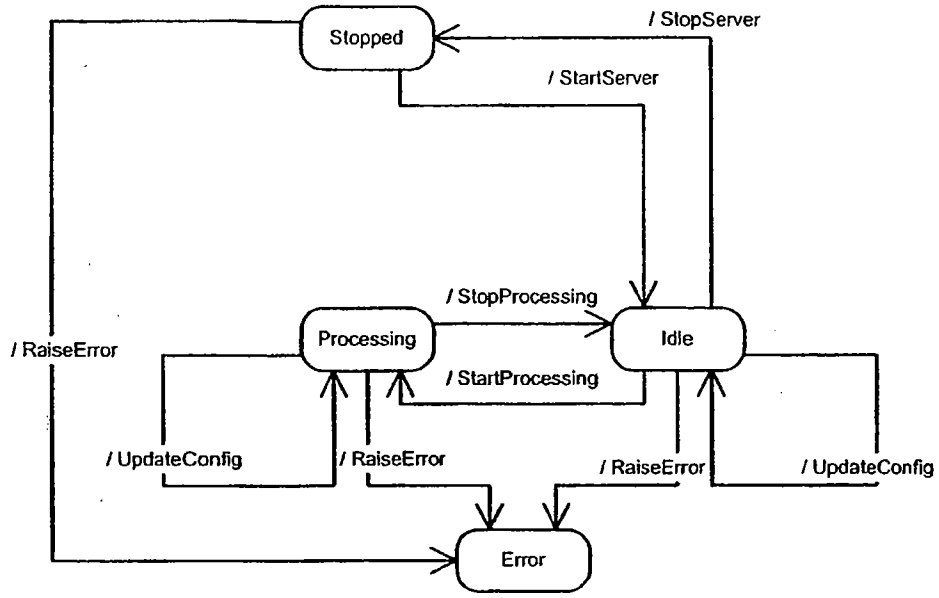
Server State Manager:

At the heart of the server component framework is the *Server State Manager*. The server state manager is a set of interfaces that initiate and manage state changes within a server component. All Server components, by virtue of being built on top of the component framework, can be managed uniformly across a deployment.

The *Server State Manager* implements a simple state machine that is shared between components. It manages the state transitions within the server component. Additionally, the state manager maintains current state information for each server component and logs state transition history in the event that a server component terminates unexpectedly.

As specified above, each server component is required to implement a number of transition methods, with pre-defined signatures, which the state manager will execute when making a state transition.

The following diagram shows the state diagram and the associated transitions:



Message Service:

The *Server Component Framework* depends on a message service which is used by the Server State Manager and Configuration Manager to communicate with the System Monitor.

The *Server State Manager* uses the messaging service to listen for state change requests from the System Monitor which it satisfies by returning the current state, any up-to-date status, and load information.

The *Configuration Manager* uses the message service to request configuration information from the *System Monitor*. Although each server component could easily go to the database for configuration information, it has been decided to go through the monitor as to save db licensing costs.

See below for more details on messaging protocols for the *Server State Manager* and the *Configuration Manager*. Also, refer to the low-level design document for details on the design of the eStream Messaging Service (EMS).

The configuration management utility is used by all server components to manage the server configurations. It provides the following functionality:

- Configuration for a server consists of a set of name – value tuples where the values themselves can be a set of name-value tuple.
- Servers can load the complete configuration from the database (indirectly).

- Servers can load the configuration for a given name.
- Servers can load the configuration from a flat file also.

On the Server Manger interface, configuration will appear as a table containing name – value tuples. The table may be hierarchical to represent nested structures containing the values which can themselves be name values. An example of a simple name-value pair would be:

port 8080

An example of nested name values would be:

Applications:
word.exe *windows2000sp3*
excel.exe *win98sp4*

On a flat file the configurations will always be name-value pairs. To represent one level nested structure the format would be:

Applications word.exe windows2000sp3
Applications excel.exe win98sp4

A common set of configurable parameters is defined for all server components. These configurations are maintained by the *Server Component Framework* in collaboration with the *Configuration Manager*. All configuration information is persistently stored within the database. The common configurations are used to initialize the server component after the component process has been launched. Refer to the configuration table below for more details on common configurations. Specialized server components can support additional configurations (non-common) depending on the server type. These configurations are read from the database and updated when a server component starts processing. They can also be updated dynamically while a server component is processing through the use of the **UpdateConfig** interface.

The list of common configurations include:

Information	Supports Dynamic Config	State	Description
ServerID	No		Unique identifier for server components. This server identity is unique within a deployment. This ServerID is not known to eStream clients. Its purpose is as a handle to uniquely identify server components.
ServerType	No		Identifies the type of server component. One of the following applies: ▪ Primary Monitor ▪ Backup Monitor

			▪ Application Server ▪ SLiM Server
DbUser	No		User name string required for database connectivity for this server ID
DbPasswd	No		Database password associated with the DbUser
Dsn	No		Data Source Name used to access the database.
PortNum	No		PortNumber used for light-weight messaging listener
MachineID	No		Machine ID is used to get at important machine information needed for all server components such as: ▪ IP address for the machine server component is hosted on ▪ Domain name for the machine ▪ Machines name
AutoReStart	Yes	Any State	Flag indicating that server component process can be restarted automatically without manual intervention
TimeOut	Yes	Any State	Specifies the timeout period for the listener. If the timeout period is reached. The component assumes that it has lost the connection. All Server components have a listener by which they receive instructions from the primary system monitor. Even the monitor has a listener that communicates with the Server Admin UI.

Command Line Utilities:

The *Command Line Utilities* component provides a consistent way to define and process command line arguments. To use this utility, the using component must define a table of arguments, which defines the valid set of arguments, whether or not they are required, and any default values.

Arguments are specified on the command line as name/value pairs. The utility implements the following command line syntax to support the name/value pairs. The argument syntax is defined as follows:

<name>=<value>

name	Name is an alpha-numeric identifier. The Name can be of arbitrary length as supported by the system however shorter names are recommended. Names are case sensitive
value	Any alpha-numeric value. Punctuation characters may also be used. Values are case sensitive

There can be no spaces between the <name>, "=", and the <value> elements. The existence of one or more spaces or tabs delineates separation between arguments on the command line.

Example: server.exe sid=267 dns=oracle user=michaelb passwd=mypasswd

- If a named argument is specified more than once on the command line, subsequent arguments will cause a diagnostic to be issued and the argument will be ignored.
- This utility allows the user to specify default values for arguments. If a default value is defined then the argument will be processed with its default in the event that the argument is not specified on the command line.
- This utility allows the user to tag specific arguments as required. If the required argument is not specified on the command line this utility will raise a diagnostic for the required argument. Not specifying a required argument will cause a fatal error.

The following options are supported:

sid	Server Component Identifier. Each server component within a deployment is uniquely identified via the sid. The sid is a handle into the database for accessing information unique to a specific server component.
dsn	Data Source Name. A data source name is necessary to establish an ODBC connection. Data Source Names are generated by an ODBC administrative tool
dbuser	User name. For database access security, all components need to connect as a specific user.
dbpasswd	password associate with the dbuser

Logging Utilities:

All servers and clients in eStream 1.0 need to log the error and access data. Logging enables component debugging and auditing support.

EStream Framework should provide logging with the following features:

- Each component will have an error and optioanally an access log file. The names of these files would be <component>_error.log and <component>_access.log.
- The files will be located in the <eStream1.0 Root Dir>\logs directory.
- The error log files will have messages with the following priorities:
  - 4-Low : A warning which can be ignored.
  - 3-Medium: A warning which needs to be looked into.
  - 2-High: Possibly Fatal Error in the component.
  - 1-Critical: Fatal Error. Needs admin assistance.
- Logging level should be configurable. The following levels are to be supported.
  - 0: Only errors will be logged. This will be the default level.
  - 1: Errors and Warnings to be logged.
  - 2: Errors, Warnings and Debugging information to be logged.
  - 3: Errors, Warnings and advanced Debugging (like memory dumps, tcp stack dumps etc) to be logged.

- Log Wrapping to be supported. The log files will wrap at a predefined size. On wrapping the following actions will occur:
  - Any existing <logfile>.bak will be deleted from the system.
  - The current <logfile> will be backed to <logfile>.bak
  - The component will continue logging to the <logfile>.

For each eStream client and server component logging the log files (component_error.log and component_access.log) should be written in eStream1.0Root\logs directory. The formats for the log files will be as follows:

Error Log:

[HEADER]
[TimeStamp] [Thread ID] [Priority] [Message]
...
[FOOTER]

An example of this log format would be:

```
**********************************************
Omnishift eStream Application Server
Server Started.
StartTim[redacted]16:31:19 -0700
IP Address: [redacted]
Logging Level: 3
**********************************************

[14/Aug/2000:16:31:19 -0700] 0 2-High Cannot connect to the database.
Invalid Username/Password.
[14/Aug/2000:16:31:19 -0700] 1 1-Critical Cannot start the HTTP listener
at port 80.
[14/Aug/2000:16:31:19 -0700] 0 1-Critical Shutting down the server.

**********************************************
Omnishift eStream Application Server
Server Stopped.
StopTime:[redacted]16:35:19 -0700
IP Address: [redacted]
Logging Level: 3
```

Format of Access Log Message:

[HEADER]
[TimeStamp] [Thread ID] [Message]

[FOOTER]

Data type definitions

Server State:

The server components can be in any one of the following states:

State	Description
STOPPED	If a server is in the STOPPED state then the component process is not running.
IDLE	Server component is up and running. The server has been initialized with the common configuration and the messaging system has been enabled. The listener is actively waiting on the System Monitor for transition requests. The server component is not processing any work specific to this servers specialization.
PROCESSING	Server component is actively taking requests and processing work specific to its specialization. ie. serving access tokens, and application file requests.
ERROR	An error has occurred in the system. Unless the server component is configured with **AutoReStart** and ERROR state must be manually cleared by the server-side administrator.

Server State Transitions:

Changes in server component state are initiated either by the *System Monitor* or directly by the server-side administrator for the system monitor. The exception to this is when an error condition is raised by a server component. In this case, the component will initiate the state change itself. The following state transitions are supported:

Action	Description
START_SERVER	Server is expected to be in the STOPPED state. If a server component is configured to support AutoReStart then the ERROR state is also a valid state from which to initiate this action.
[illegible]	[illegible] from any state.
START_PROCESSING	Causes the server to change from the IDLE state to the PROCESSING state.
STOP_PROCESSING	Causes the server to change from processing to IDLE state.
UPDATE_CONFIG	Request that the server read its configuration from the configuration manager and change its configuration.

RAISE_ERROR	Request that the server go to ERROR state. This causes an error handler to be called. If the error is fatal it will cause immediate termination of the server process.

Finite State Table:

```
FSMTableEntry ServerStateMgr::FSMTable[] =
{
        { START,{{START_SERVER,STOPPED, START_SERVER,NULL},
                {START_PROCESSING, STOPPED ,START_PROCESSING, NULL},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {STOPPED, {{START_SERVER, IDLE, NULL_REQUEST, &StartServer},
                {START_PROCESSING, IDLE, START_PROCESSING, &StartServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {IDLE,{{START_PROCESSING, PROCESSING, NULL_REQUEST,
&StartProcessing},
                {STOP_SERVER, STOPPED, NULL_REQUEST, &StopServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {UPDATE_CONFIG, IDLE, NULL_REQUEST, &UpdateConfig},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {PROCESSING,{{STOP_PROCESSING, IDLE, NULL_REQUEST,
&StopProcessing},
                {UPDATE_CONFIG,PROCESSING, NULL_REQUEST,
&UpdateConfig},
                {STOP_SERVER, IDLE, STOP_SERVER,  &StopProcessing},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        { ERROR,{{STOP_SERVER, STOPPED,NULL_REQUEST, NULL},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} }

        {NULL_STATE,{{NULL_REQUEST, NULL_STATE, NULL_REQUEST,
        NULL} } }
};
```

Messaging Service Protocol:

A light-weight messaging protocol is needed to facilitate communication between server components. The primary purpose of the messaging protocol is to communicate transition requests to the server components. In response, server components communicate state, status, and load information back to the *System Monitor*.

The messaging protocol supports two primary message types. 1) Requests for the *System Monitor* to perform on other servers. 2) Requests to the server components themselves. These message types are distinguished through the protocol as described below. If the receiver ID and the target ID are identical then the request is for the receiver. If the target is different than the receiver, the message is for the *System Monitor* to enact a request on the target component.

All requests are required to be acknowledged. Without an acknowledgement the message is considered un-received.

OpCode	senderID	receiverID	targetID	Data

The following table describes the protocol used by the Server State Manager in its communication with the System Monitor.

OpCode	Description	Data
0x01	Request for current state	None
0x02	Acknowledgment	▪ Current state ▪ Load info ▪ Status info
0x03	Stop Server request. Acknowledged with 0x02 message	None
0x04	Start Server request. Only valid for System Monitor. Acknowledged with 0x02	None
0x05	Start Processing Request. Acknowledged with 0x02	None
0x06	Stop Processing Request. Acknowledged with 0x02	None
0x07	Update Configuration Request. This is a request for a server component to request its specialized configuration information from the System Monitor and update itself. Acknowledged with 0x02.	None

The messaging protocol used by the configuration manager is described below:

OpCode	Description	Data
0x01	Request a complete reload of Configuration	
	items.	being requested. ▪ array of names of configuration items.
0x03	Acknowledgement of configuration reload request.	▪ Number of tuples being returned ▪ Flat representation of configuration tu-

		ples

Interface definitions

Server State Manager:

```
class ServerStateMgr
{
private:
        ServerState CurrentState;
        static FSMTableEntry FSMTable[];

public:
        ServerStateMgr(void);
        ~ServerStateMgr(void);

        ServerState SetState(ServerState);
        ServerState GetState(void);
        ServerState ProcessRequest(ServerRequest);
};
```

SetState	**Description:** Sets the current state of the server component. 1. Log the state change request 2. Update the state field within the server component in memory data structures. 3. Send message to requester informing them of the successful state change. Note: **SetState** does not update the database directly as in the orginal design. The database is updated by the *System Monitor* once it has received an acknowledgement. A state transition is not complete until **SetState** returns successfully and the Monitor has update the database. **Input:** state value to set current state to. **Output:** current state after the new value has been set. If an error occurs will go to error state. **Errors:** 1. Invalid state argument 2. Failure to either connect or commit state change to the database.

GetState	**Description:** returns the current state. This function does not read from the nent is up and running and that it maintains a valid state. **Input:** none. **Output:** returns the current state. **Errors:** None. Will always return a valid state.

ProcessRequest	**Description:** request to the Server State Manager to change server state. This routine implements the guts of the state machine.

	1. Get the current state, and transition request 2. Index into the FSM table and continue to transition from state to state until the transition request is satisfied. 3. Each state transition calls the specialized transition routines for each component. 4. Call to **SetState** to complete each state transition. 5. In the case of an error the state machine will process a RAISE_ERROR request which will call the specialized **HandleError** and transition to the ERROR state. **Input:** server transition request. Refer to table of valid requests defined above. **Output:** current state after the request has been completed. **Errors:**

Server Component Framework:

```
class ServerComponent: ServerStateMgr{    // abstract base class
private:
        ErrorInfo*      Error; // maintains error if error was detected
        ServerConfig*   Config;  // holds common configuration
        Connection*     Listener; // messaging        utility
public:
        virtual int StartServer(void); // may be specialized by a server component
        virtual int StopServer(void); // may be specialized
        virtual int StartProcessing(void) = 0; // must be specialized
        virtual int StopProcessing(void) = 0; // must be specialized
        virtual int UpdateConfig(void) = 0; // must be specialized
        virtual int HandleError(void) = 0; // must be specialized
        void Run(Request);
}
```

StartServer	**Description:** Called by the *Server State Manager* when a server component is to be started. The **StartServer** routine is provided as part of the SeverComponent class. It performs the following: 1. Send request to System Monitor to request an update of common configuration information. 2. Apply the configuration information to the server component. 3. Construct a listener connection object and start the message service. 4. Return success or failure. Note: ▪ This routine must return immediately to the main thread. Otherwise the *Server State Manager* will be blocked. ▪ Successful return from the **StartServer** routine will put the server into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error condition

StopServer	**Description:** Called by the *Server State Manager.* 1. Perform any necessary cleanup. 2. Send last acknowledgment confirming shutdown to requester 3. Shut down the messaging system and the listener. 4. exit process Note: The monitor will update the database and perform logging. **Input:** None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error value

StartProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by each specialized server component. This routine is used to provide all functionality unique to different types of servers. 1. Spawn a primary processing thread (also known as the boss thread). a. Read server specific configurations unique to this type of server component from the System Monitor b. Spawn worker threads. Depending on the server type this routine does the heavy lifting to either process access tokens and renewals in the case of SLiM server, or process file requests for application servers, or manage and monitor the server components in the case of the *System Monitor.* Note: ▪ This routine must return immediately so that the *Server State Manager* can continue to operate in the main thread. ▪ This routine may make use of the *Server Configuration Manager* for obtaining specialized configuration information **Input:** None **Output:** Value of 0 if successful else error condition. **Errors:** TBD

StopProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by the specialized server component type. 1. Reverse all actions performed by the **StartProcessing** routine. All worker threads should be joined or pooled in waiting state. Successful return from this routine will put the server component into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

UpdateConfig	**Description:** Called by the *Server State Manager*. This routine must be defined by the specific server component type. The purpose of this routine is apply dynamic configurations or update specialized configurations that are unique to this server component. <may require adding a new state to separate dynamic and static configurations> Input: None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

HandleError	**Description:** Component defined error handling routine to handle errors such as timeouts, etc. This routine will need to handle a number of error cases as are possible by the specialized component. The error information is maintained with

	the ServerComponent class. **Input:** None. **Output:** Integer value designating a handled error or failure. If the error cannot be handled then it is fatal. **Errors:** TBD

Run	**Description:** This routine implements the main processing loop for the server component and runs in the main thread. This routine drives the server component by initiating state requests from the *System Monitor*. Note: The *Server State Manager* always runs in the main thread. 1. Call **ProcessRequest** to transition the server component into the initially requested state. 2. Enter main processing loop a. Check for requests from the message service. b. Call **ProcessRequest** to service the request. c. Send acknowledgement for the request to the message service. Acknowledgement includes new state, load info, and status. **Input:** Initial Transition Request **Output:** None. This routine should never return **Errors:** None.

Server Component Main Loop:

The following main loop is common to all server components:

```
void ServerComponent::Run(ServerRequest Request)
{
        ProcessRequest(Request);
        while (1)
        {
                Request = Listener->GetRequest();
                ProcessRequest(Request);
                Listener->AckRequest(Request, GetState, GetLoad, GetStatus);

        }
}
```

```
#include "ServerArgs.h"
#include "Server.h"

int main(int argc, char* argv[]) {
        Args = new ArgList();
        Args->ProcessArgList(argv, argc);
        Server = new ServerComponent(GetValue(SID),
```

```
                                        GetValue(DNS),
                                        GetValue(DBUSER),
                                        GetValue(PASSWD));
        Server->Run(START_PROCESSING);
}
```

Command Line Utilities:

```
class NameValuePair
{
        private:
                char* Name;
                char* Value;
        public:
                NameValuePair();
                ~NameValuePair();
                char* GetValue(void);
                char* GetName(void);
                char* SetName(char*);
                char*SetValue(char*);
};
```

```
typedef int (*pFunc)(NameValuePair*);

struct ArgTblEntry
{
        char*   Name;
        bool    Required;
        char*   DefaultValue;
        pFunc   ProcessFunction;
};
```

```
ArgTblEntry const ServerArgsTbl[] = {
        {"sid",         true,   0,      &ProcessSId},
        {"dsn",         true,   0,      &ProcessDsn},
        {"dbuser",      true,   0,      &ProcessDbUser},
        {"dbpasswd",    true,   0,      &ProcessDbPasswd},
        {0              0       0       0}
};
```

```
typedef vector<NameValuePair*>    ArgVector;

class ArgList
{
        private:
```

```
        ArgVector               ArgVec;
        const ArgTblEntry*      ArgTbl;

    private:
        NameValuePair*          ParseArg(char* Arg);
        char*                   ParseName(char* Arg);
        char*                   ParseValue(char* Arg);
        int                     ProcessArg(NameValuePair*);
        int                     FinalizeArgs(void);

    public:
                                ArgList(const ArgTblEntry*);
        int                     ProcessArgList(char* argv[], int argc);
};
```

ProcessArgList	**Description:** Process the entire argument list. In a loop for each argument argv[] … 1. Call **ParseArg** passing in argv[]. 2. **ParseArg** passes the result to **ProcessArg** 3. After processing the entire argument list and exiting the loop call **FinalizeArgs** **Input:** argv and argc as passed into main() entry point **Output:** integer value designating success or failure **Error:**
ParseArg	**Description:** Takes a char* argument and verifies that it follows that name/value syntax defined as <name>=<value> **Input:** Next char* argument on the list **Output:** **NameValuePair**. NULL will be returned in the event of a syntax error **Error:**
ProcessArg	**Description:** This routine performs the semantic analysis of an argument. 1. Look up name in the **ArgTbl** 2. Verify that the value is valid 3. Add the name value pair to a list of processed arguments called **ArgVec** list. 4. If this name value pair already exists in the list then issue a diagnostic. 5. Call the supplied processing function for this argument as specified in the **ArgTbl** **Input:** **NameValuePair** **Output:** Interger value designating success or failure (0 for success, positive integer for other errors) **Error:**
ParseName	**Description:** Verify that the Name part of the argument conforms to being alpha-numeric **Input:** char* Name part of argument **Output:** char* Name else NULL **Error:** None
ParseValue	**Description:** Verify that the Value part of the argument conforms to being alpha-numeric and/or punctuation characters **Input:** char* Value part of argument **Output:** char* Value else NULL **Error:** None
FinalizeArgs	**Description:** Post process the argument list. The purpose of this routine is to validate that all required arguments have been defined on the command line. Also processes and adds default arguments to the ArgVec. **Input:** None **Output:** Success or Failure **Error:**

Configuration Manager:

```
class  Tuple {
        string name;
        Value value;
};

class Value {
        int type;
};

class StringValue: public Value{
        string value;
};

class TupleValue: public Value {
        vector <tuple> tupleArray;
};

typedef vector < tuple > ConfigArray;

class ServerConfig {
private:
        ConfigArray Array;
public:
        ServerConfig(serverId, dns, dbuser, dbpasswd); // Initialize from db
        ServerConfig(serverId, string filename); // To initialize from a file.

        ConfigArray* GetConfigArray(int serverId);
        Tuple* FindConfig(string Name);
        int Reload(void);
        Tuple* GetConfig(int serverId ,string Name);
};
```

ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes configuration manager. 2. Opens the database and gets configuration array **Input:** Server Id, Data Source Name, Database User name, and database users password. **Output:** None **Errors:**
[illegible]	**Description:** Constructor for Configuration Manager. 1. Initializes Configuration Manager. 2. Opens configuration file and reads configuration array. **Input:** filename of flat-file configuration. **Output:** None **Errors:**

GetConfigArray	**Description:** Returns the entire configuration for a given server id. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying which server to retrieve configuration for **Output:** Returns a vector holding the configuration or NULL **Errors:**

GetConfig	**Description:** Returns the configuration for the specified name. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying the server to retrieve configuration for and Name of configuration item. **Output:** Configuration Tuple. A Tuple may be a nested Tuple. NULL if an error is encountered. **Errors:**

FindConfig	**Description:** Returns the Tuple specified by the name. This routine does not go to the database or flat-file to get its value. Rather it finds the value in the ConfigArray maintained by the Configuration Manager. **Input:** Name of the configuration item. **Output:** Configuration Tuple. NULL if an error is encountered or the Tuple does not exist in the current configuration. **Errors:**

Reload	**Description:** Reloads the entire configuration from the database or flat-file. This routine may reload its configuration indirectly through the use of the System Monitor. In this case it will make a message request to the monitor and listen for the configuration results. **Input:** None **Output:** integer specifying success or failure. Zero will be returned in the case of Success. A negative value in case of error. **Errors:**

Logging Utilities:

```
class LogManager
{
private:
        char* FileName;
        int MaxFileSize;
        char* ResourceFile; // message catalog file

        char* GetMessage(MsgNum, MsgStr)
public:
        LogManager(ServerId,Size=10);
        LogMessage(MsgStr);
        LogMessage(ThreadId, MsgNum, MsgStr, ...);
```

```
};
```

LogMessage	**Description:** Write message out to log file. There are two forms of LogMessage. The first will write out a message buffer as is (unformatted) bypassing the resource file. The second form will format the message. Both forms of LogMessage always pre-append a time stamp. 1. Lookup message number in the resource file and get message string 2. format the log message using time stamp, thread id, etc. 3. write out message into the log file. **Input:** Thread Id, Message Number, Message String, and variable number of arguments. **Output:** None. **Error:**

GetMessage	**Description:** Routine returns a message string from the resource file for the message number specified. **Input:** Message number, C Locale text string. **Output:** Message string. Either way, Get Message will always pass a return a valid message string by either returning the string from the resource file or by passing back the MsgStr passed in. **Error:** If an error occurs trying to get a message from the resource file, a message will be logged to the error log.

```
class ErrorLog: protected LogManager
{
private:
        LogLevel ErrorLogLevel;
public:
        ErrorLog(ServerId, LogLevel=0, Size=10);
        LogError(ThreadId, ErrorNum, ErrorMsgStr, ...);
};
```

LogError	**Description:** Writes output to error log file. 1. Check that the message level against the current ErrorLogLevel. 2. Format the message and call the long form of **LogMessage** to write the buffer out to the file. **Input:** 1. ThreadId: Thread identifier to help with the debugging process. 2. ErrorNum: Error number used to uniquely identify an error message in the resource file. 3. ErrorMsgStr: Message string which includes stdio like string formatting. 4. ...: variable list of arguments to be inserted into the message string per the format. **Output:** None. **Error:**

Testing design

Each of the components that make up the Server Component Framework will be able to be tested independently of the other components. Each component will have a main entry point defined within a testing .exe to accomplish the Unit testing phase.

Testing of the component framework will be done in phases. Each of the phases is described below along with its dependencies.

Phase 1: Unit testing Test basic components that make up the framework. Each components functionality, restrictions, and boundary conditions will be tested. Will allow testing common configurations for a single server component. This round of unit testing will test the integrated component utilities and framework.	**Dependencies:** 1. ServerComponent class 2. ServerStateMgr class 3. ArgList class 4. Logging Utilities 5. Configuration Manager (flat-file)
Phase 2: Unit testing (full functionality) Test full functionality including messaging interfaces and database connectivity.	**Dependencies:** 1. Phase 1 2. Database connectivity 3. Messaging Service
Phase 3: Integration Testing	**Dependencies:** 1. Phase 2 backup) 3. SLiM Server, App Server, Web-Server
Phase 4: Stress Testing See section on stress testing for details	**Dependencies:** 1. Phase 3

Unit testing plans

Command Line Utilities

The Command line utilities will be tested in a stand-alone module called cmdline.exe. It will support the command line arguments defined in this document.

Configuration Manager

The configuration module is a stand-alone module which will be tested using a config-test.exe executable. The executable will exercise all of the interfaces described above. The configtest.exe executable should be testable in the DB and the non-DB mode.

Logging Utilities

The logging utility will be built as a DLL (otlog.dll). We will provide a binary otlog-test.exe which will exercise each of the interfaces mentioned above.

Server State Manager

The Server State Manager and the Server Component Framework will be tested independently of specialized components. The routines that require specialization (**StartProcessing**, **StopProcessing**, **HandleError** and **UpdateConfig**) will be provided to simply return successfully.

Stress testing plans

Stress testing will require having at least the System Monitor functionality implemented since it is used to drive the server components.

1. Test to repeatedly start, stop, reconfigure the server component.
2. Test to crash machines with server components to validate:
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
3. Test to kill individual server component processes.
    a. data persistence.
    b. detection capabilities and response.
4. Test lost database connectivity
5. Test lost of messaging capabilities
    a. repeatedly losing and re-establishing messaging connectivity
6. Test error recovery under adverse conditions.
7. Test recovery from running out of memory, thread resources.
8. Test recovery from threads dying.
9. etc.

Coverage testing plans

1. Goal: 100% path flow coverage. Only exceptions for known error conditions that cannot be practically reached (e.g. thread synchronization, etc.)

Cross-component testing plans

The following pair-wise testing will be performed:

1. framework/database (phase 2)
2. framework/messaging (phase 2)
3. framework (System Monitor) /framework (backup Monitor) (phase 3)
4. framework/Web Server (phase 3)
5. framework (System Monitor) /framework (Other Servers) (phase 3)

Upgrading/Supportability/Deployment design

1. Each error condition will be documented with explanations and practical work-arounds
2. Component framework will support enhanced debug option to dump additional de-bugging information to special log files.

Open Issues

eStream System Monitor Low Level Design

Michael Beckmann
[redacted]

Functionality

The role of the System Monitor is to monitor the state of the Application Servers and SLiM servers within an eStream deployment. In addition, it also manages a back-up System Monitor.

- The System Monitor provides the following key services:
    a. Monitors and reports server load across machines
    b. Monitors server state across machines.
    c. Acts as a communication conduit to the database for configuration information needed by the server components.
    d. Initiates state changes within the server.
        i. Start/Stop servers
        ii. Sends requests for servers to update their configurations
- The system monitor runs as it's own process. Within a multi-system deployment, there will be at least two monitoring processes, each on a different machine:
    a. One monitoring process will act as a primary and the others as backups.
    b. In the event that the primary monitor goes down, one of the backups will take over the primary monitoring responsibilities.
- The monitoring process can re-launch a server side process if a process terminates unexpectedly (this is a configurable option).
- The system monitor manages server state by maintaining regular communication via a heart beat protocol between itself, the backup monitors, and every logical server.
- The monitor will raise an alarm if it does not receive a heart beat response from the servers within a specified period of time.
- The system monitor's heart beat request rate is a dynamically configurable parameter maintained within the database.
    a. The rate can be changed through the administrative interface.
- The system monitor's heart beat supports a light-weight messaging protocol between components to initiate state changes.
    a. Simple request pulse.
    b. Stop request pulse
    c. Configuration request pulse

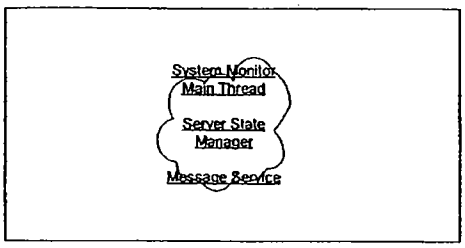
The System Monitor process is extended from *the Server Component Framework* by implementing the plug-able interfaces which includes:

StartProcessing

StopProcessing
UpdateConfig
HandleError

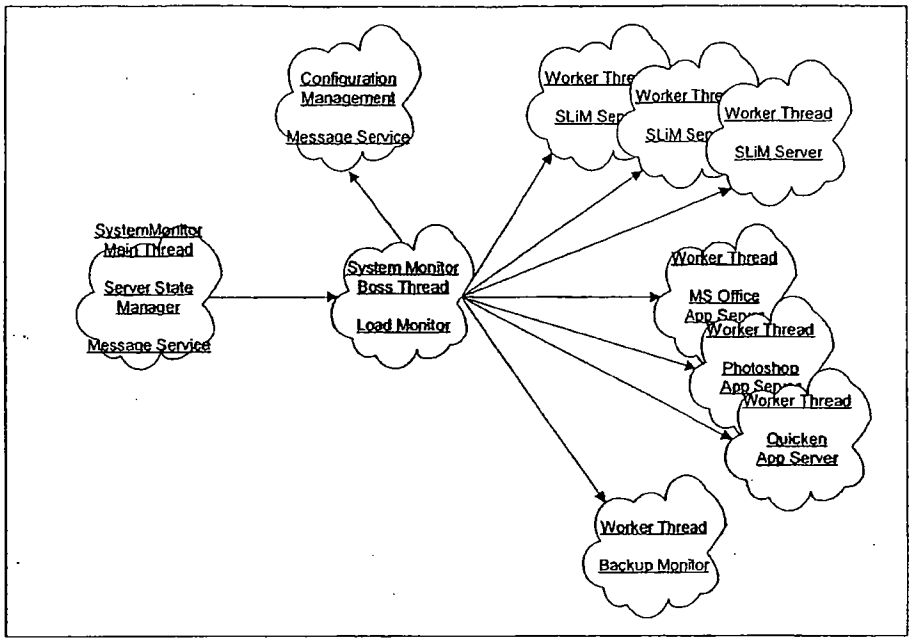
System Monitor States:

When the System Monitor transitions from the STOPPED state to the IDLE state it is maintaining only its main thread which it inherits from the Server Component Framework and runs the Server State Manager. The messaging Service may or may not maintain its own threads. For the purpose of this design we will assume that it is transparent to the System Monitor.



When the System Monitor transitions into the PROCESSING state it employs a "Boss-Worker" parallel programming model:

1. The System Monitor creates a processing thread that acts as the boss thread.
2. The boss thread reads the configuration for all server components and spawns worker threads for each logical server component.
3. Worker threads are allocated, in turn, by the boss to start, monitor, and manage each server process running within a deployment.
4. The boss thread spawns a special thread to handle configuration requests from all the server components.
5. The *Load Monitor* runs as a service within the boss thread.



Synchronization between worker threads and the boss thread is provided through request queues.

Load Monitor:

The *Load Monitor's* role is to aggregate the load information for each server component within the deployment and update aggregated information to the database.

The *Load Monitor* runs in the "boss" processing thread of the *System Monitor*.

For each application the *Load Monitor* maintains a list of servers and their response time. It periodically updates the database with the order lists which are consumed by the SLiM servers. The frequency at which it updates the database is configurable.

The *Load Monitor* maintains three interfaces:

UpdateLoad	Update in memory server sets with the current load information passed in as an argument. This routine is called by the boss thread when the System Monitor is in the PROCESSING state.
DbGetServerSets	Retrieve current server sets in the event that System Monitor crashes and/or starting System Monitor.
DbSetServerSets	Update database with lists of servers for each app

Data type definitions

The System Monitor maintains a few key data structures beyond what is provided by the *Server Component Framework*:

Request Queue:

Each worker thread will maintain its own mutex protected input queue. The input queues are used primarily to communicate requests between threads. A queue entry models, fairly closely, the data contained within the messaging protocol between servers as defined in the *Server Component Framework*. This data can be used for the input queues for each worker thread managing individual work components.

The details of the input queues themselves is defined in the common thread API.

```
struct QueueEntry {
        int Request
        ServerID SenderID
        ServerID RecieverID
        ServerID TargetID
        Data
}
```

Managing Worker Threads:

The System Monitor's boss thread maintains an in memory list of all known servers within a deployment, and their associated worker thread ID. In addition, it maintains information about request queues for each worker thread. This list is maintained as the **SystemMonitorList**. The list is made up of entries which maintains all the necessary information to start/stop/manage/communicate/etc. with a server component.

```
SystemMonitorEntry = struct {
        ServerConfig* ServerInfo;
        Thread ThreadID;
        ThreadQueue* InputQueue;
}

typedef vector <SystemMonitorEntry> SystemMonitorList;
```

Interface definitions

The implementation of the System Monitor is derived from the generic *ServerComponent* class. Refer to the *Server Component Framework* Low Level Design.

Implementing a *SeverComponent* requires the definition of the following methods:

StartServer	**Description:** This routine is called to start a server process and bring it to the IDLE state. It assumes that the server is in the STOPPED state. This routine performs all the necessary initialization and configuration common to all server components. This function is predefined in the *ServerComponent* base class. The System Monitor has no need to override it. **Input:** None. **Output:** Integer value designating return status. Success = 0. **Errors:**

StopServer	**Description:** The routine StopServer is called to cleanup and terminate a server process. It assumes that the server component is in its IDLE state and is therefore not processing any requests. This function is predefined in the *ServerComponent* base class. The System Monitor has no need to override it. **Input:** None. **Output:** Integer value designating return status. Success = 0. **Errors:**

StartProcessing	**Description:** This routine initiates all the activities unique to the System Monitor: 1. Launches a boss thread. Call **MonitorServers** as the boss thread entry point. 2. Return immediately Note: The **StartProcessing** routine must return immediately else the Server State Manager will be blocked. **Input:** None. **Output:** Integer value designating return status. **Errors:**

StopProcessing	**Description:** This routine will perform the following activities: 1. Terminates each server component's monitoring thread. This effectively disables the monitor from managing any executing server components including stopping existing components or starting new ones or servicing configuration requests. 2. Disables the load balance manager. Note: The primary system monitor will only execute this interface if a system administrator explicitly changes the monitors state or an error has occurred within the monitor. **Input:** None. **Output:** Integer value designating return status. **Errors:**

UpdateConfig	**Description:** This routine will perform configuration changes specific to system monitor functionality. **Input:** None. **Output:** **Errors:**

MonitorServers	**Description:** This routine is the thread entry point for the boss thread. It is the primary thread for managing and monitoring all the server components. The *Load Manager* runs in the boss thread. 1. Launch the configuration management within its own thread. 2. Construct/initialize *Load Manager* 3. Go to the database and get a list of all the server components. 4. For each server component retrieve its common configurations and create and entry into the **SystemMonitorList** 5. Spawn a worker thread to manage/monitor the backup System Monitor. Call **MonitorServer** as the thread entry point passing in the **SystemMonitorEntry** After the backup monitor has been launched, repeat step 4 for each server component. 6. Loop back and redo steps 2-5 looking for new servers to manage **Input:** **Output:** **Errors:**

MonitorServer	**Description:** Launches and monitors the specified server component process. It can launch server process' locally and on remote machines. It is expected to be the start routine for a new thread. Performs the following steps: 1. Call **StartServerProcess** to launch the server component per the **SystemMonitorEntry**. 2. Open a point-to-point messaging connection to the newly launched server component. 3. Initiate a regular heart-beat request. 4. Enter monitoring loop a. Check for requests from the worker thread input queue b. Process requests by sending request to the server currently being monitored by this thread. c. Listen and wait for response; timeout if wait is too long. If there is a timeout exit thread with an error and let the error handler deal with it. d. If a response is received update state information in **SystemMonitorEntry**. e. Enqueue load information onto Load Balance managers input queue. f. repeat a-f **Input:** **SystemMonitorEntry**. **Output:** Status of the server component on exit. **Errors:** 1. launch error 2. messaging error 3. unexpected server component termination

Primary and Backup System Monitor:

The backup System Monitor needs to be able to take over the primary system monitor responsibilities in the event that the backup loses communication with the primary. The following interface will cause the backup monitor to take over as primary.

SwitchPrimaryMonitor	**Description:** Take over primary monitor responsibilities. 1. Update database validating monitor switch. 2. Shut down the old primary just in case it is still running using by opening a connection to the old primary and sending it a STOP_SERVER request. 3. Go through the same steps as the primary monitor would do when its **StartProcessing** routine is called including starting up a backup system monitor Note: It may be more appropriate to define an error handler that is called when the heart beat is lost. The error handler would go through steps 1 and 2 and then request a state change which would result in the monitor coming on line as a primary monitor. This requires more thought but would be a more elegant solution. **Input:** None **Output:** integer value designating success or failure **Error:**

StartServerProcess	**Description:** This routine spawns the requested process either on the local machine or on a remote machine. 1. Verifies that the target machine is up and running. 2. Launch process specified according to attributes **Input:** Process name, target machine, attributes **Output:** integer return where 0 designates success else error code is returned **Errors:** 1. machine not responding 2. executable not found 3. failure on launch

Component design

Testing design

The System Monitor will provide a command line option to facilitate testing the component in its various testing phases. The option will allow the system to manage its

Unit testing plans

The System Monitor can be unit tested in the following phases:

Phase 1: 1. Test state management and messaging capabilities of all server components 2. Test starting, stopping, updating configs	**Dependencies:** 1. Common Server Framework 2. Messaging
Phase 2: 1. Test data persistence and error recovery 2. Test config updates from database 3. Test back-up monitor and switch-over	**Dependencies:** 1. Phase 1 2. Database Connectivity
Phase 3: 1. Test load balance manager and components sending load information	**Dependencies:** 1. Phase 1 & 2 2. Load Balance Manager
Phase 4: 1. Full functionality testing 2. Stress testing 3. Full error recovery testing	**Dependencies:** 1. Phase 1, 2, & 3 2. Web Server 3. App Server/Slim Server

Stress testing plans

Stress testing will be performed at Phase 4 testing. Tests should include:

1. Max # of generic server components on a single machine.
2. Max # of generic components across multiple machines.
3. Max # of generic servers changing state at least once per second
4. Test to repeatedly start, stop, reconfigure each server component.
5. Test to crash machines with server components to validate:
    a. data persistence.
    b. detection capabilities and response.
    c. Auto Restart.
6. Test to kill individual server component processes.
    a. data persistence.
    b. detection capabilities and response.
    c. Auto Restart.
7. Test lost database connectivity
8. Test lost of messaging capabilities
    a. repeatedly losing and re-establishing messaging connectivity
9. Test error recovery under adverse conditions.
10. Test recovery from running out of memory, thread resources.
11. Test recovery from threads dying

Coverage testing plans

The System Monitor will achieve 100% code coverage with the exception of error conditions which are possible however difficult to reach in practice.

Cross-component testing plans

The System Monitor interacts with the following components:

1. System Monitor/Database
2. System Monitor/WebServer
3. System Monitor/Server Component Framework (other servers)

Upgrading/Supportability/Deployment design

1. All diagnostics will be documented as to their root cause and workarounds/actions to be taken.
2. The system monitor will support an enhanced debug support which dumps additional information to special debug logs.

Open Issues

1. Need to figure out how to launch a process on a remote machine. May need to use the slave monitor to actually launch the process. But then the question arises ... how does one launch the backup monitor?
2. Need to identify a mechanism to ensure that we do not have more than one primary monitor running at any given time.
3. Need to strike a balance on how many threads to spawn. All the server components may be able to be monitored and managed out of a single primary thread which is referred to as the boss thread.

eStream Web Server/Database Low Level Design

Bhaven Avalani

[redacted]

Functionality

The eStream solution provides a set of account, user, and subscription management utilities. These utilities are provided as extensions to the ASP's(Application Service Provider) web server.

There are three categories of users for these utilities: End User, Group Administrator and ASP Administrator. The roles and the capabilities of each of these users are detailed below.

End user for a system is the user who will actually access eStream application using the eStream clients. An end user should be able to:

- Create Account and User attributes. (Username, Password, etc.)
- Change Account and User attributes.
- View all available applications in the eStream system.
- Subscribe/Manage eStream applications.
- View Account Status.
    1. List of applications subscribed.
    2. Status of current subscription.
    3. View/Change Billing information.
    4. View/Change Account information.

A *Group Administrator* is an administrator for a group of users. An individual user is by definition a group administrator for a single user group. Capabilities of a group administrator are:

- (All of single user capabilities).
- Add delete users from a group.
- Manage the active sessions for a group. A group manager should be able
- View the billing information. This will probably need hooks to an external billing system.

An *ASP administrator* manages the overall application system. Capabilities of an ASP administrator are:

- Manage accounts/users/subscription for all users/groups in the system.
- Manage the application data for a subscription system.

1. Add new applications to the system.
2. Modify application information for the system.
3. Provide the pricing mechanism for the applications(?).

- Manage the servers in the system.
  1. Configure a server.
  2. Stop/Start a server. This is accomplished by a message to the Monitor server.
  3. Get load information for a server.
  4. Get logging information for a server.

There are essentially two different types of accounts, which the system will support: Single user account and corporate accounts.

The following licensing mechanisms will be supported by the system.

- Fixed Duration License. (Typically monthly license).
- Fixed Duration Floating License. An example of this is n licenses for k users for a fixed duration.
- Indefinite License.

Description

There are several key issues that need to be determined for the Web Server architecture. The options available in the market to implement these technologies are listed below.

Web Server:

- Apache
- Netscape Server
- Microsoft Internet Information Server

CGI Technology

- Servlet/JSP
  - Tomcat (from Apache group)
  - JRun (from Allaire)
- Active Server Pages (available on NT only)
- NSAPI (C level API available for Netscape and Apache).
- ISAPI (C level API available for IIS and Apache)
- CGI (Perl/C etc.).

Database Connectivity

- JDBC.

- ODBC
- Native.

Database

- SQLServer
- Oracle
- Sybase
- Informix
- LDAP(??)

The overall proposed solution for eStream 1.0 WebServer release is:

Apache + Tomcat(for JSP/Servlet) + JDBC + SQLServer.

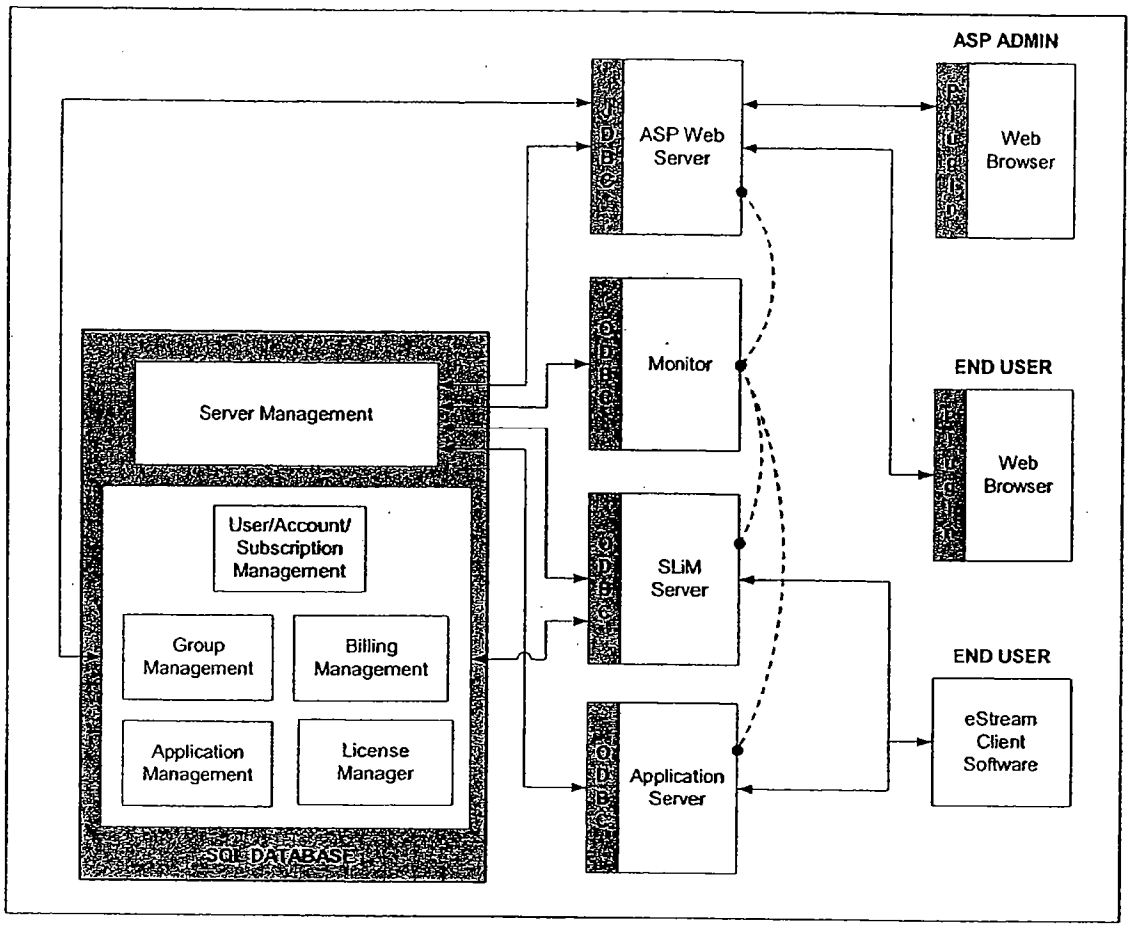
The reasons for choosing this combination for the servers are as follows:

1. JSP/Servlet is the only technology which is available for cross-platform and cross WebServer support.
2. We need to decide on a single web server to develop and test against for release 1.0. Apache is chosen to be the one as it is popular on Unix and NT platforms and it is freely available.
3. Tomcat(Apache group's reference implementation for JSP/Servlet specs) is the preferred CGI technology as it works well with Apache and all other web servers.
4. JDBC is preferred for database connectivity as its database neutral and works well with Java environment of Servlets.
5. SQLServer is the preferred database for release 1.0. This contains the scope for testing and deployment for eStream 1.0.

Since all other servers(App Server, SLM Server and Monitor) are C++ components, the following technology combination will be available for Database Access.

ODBC + SQLServer.

The data model for the eStream 1.0 database essentially consists of two high level components. The database deployment architecture is shown below:



Server Management Component: This component's primary responsibility is to manage the configuration, load and log information for a logical server in the system. The clients to this component are all the servers and administration manager. A detailed list of interfaces for this component is described in the interfaces section.

User/Account/Subscription Management: This component is responsible for maintaining the user account and subscription information for the system. The end user using the end user interface performs the updates to this component. Slim Server will access this component to validate subscriptions. A detailed list of interfaces for this component is described in the interfaces section.

Group Management: This component is useful for managing groups of users. The group administrator can only perform updates to this component. A detailed list of interfaces for this component is described in the interfaces section.

Billing Management: This component's responsibility is to provide interfaces to an external billing system. A detailed list of interfaces for this component is described in the interfaces section.